The University of Alberta

Implementation of Consistency Techniques in WUP

by

Siu-Lung Gordon Fong

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall 1989

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Siu-Lung Gordon Fong

TITLE OF THESIS: Implementation of Consistency Techniques in WUP

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1989

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .....

Permanent Address:
65B San Hong St., 2/F,
Sheung Shui, N. T.,
Hong Kong

Date: Sept. 6, 1989

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Implementation of Consistency Techniques in WUP** submitted by **Siu-Lung Gordon Fong** in partial fulfillment of the requirements for the degree of Master of Science.

Tian Huai You
(Supervisor)

Roger W Toogood

Date: Sept. 6, 1994

To My Parents

Acknowledgements

I would like to thank my supervisor Dr. Jia You for his guidance and assistance throughout the research and the preparation of this thesis. His patience and kindness are greatly appreicated.

I would also like to thank the members of my examining committee Dr. R. Goebel, Dr. H.J. Hoover, and Dr. R.W. Toogood for the time they spent in reviewing this thesis.

Financial support from the Department of Computing Science makes my stay in Edmonton possible.

Mr. Alan H.W. Kwan and Mr. Charles Chirchir had proofread most of this thesis. Their comments and encouragement mean much to me.

I would like to thank members of the Paul Fellowship for their prayers, encouragement and moral support.

Lastly, many friends had in various ways assisted me to finish this thesis: Vickitt Lau, Brian Wong, Kris Ng, C.S. Law, Hang Du, Keith Fenske, Anthony Ng, Patrick Lau.

Abstract

Logic programming languages offer appealing features for solving constraint satisfaction problems (CSPs); their relational form and support of non-determinism allow CSPs to be expressed naturally as logic programs. Yet this natural expression often leads to the use of inefficient search strategies. So far, logic programming languages are seldom used to solve real-life CSPs. In contrast, many consistency techniques (CTs) are developed to solve CSPs efficiently. In these techniques, constraints can be used actively to prune the search space in an *a priori* fashion. That is, combinations of values that cannot appear in a solution are removed as early as possible so that failures are largely avoided. Therefore, a viable solution to solve CSPs efficiently in a natural formulation is to embed CTs into logic programming.

CHIP (Constraint Handling in Prolog) is the first logic programming language that supports CTs. It can solve real-life CSPs with efficiency comparable to that of some specially coded programs. On the other hand, this language is built around a modified Mu-Prolog which (in turn) relies heavily on a sophisticated delay mechanism not found in most conventional Prolog systems. The question of how to incorporate CTs into these systems is still much unanswered. This research tries to tackle this problem with a particular example; we have incorporated CTs into a conventional Prolog interpreter — the Waterloo UNIX† Prolog(WUP). This thesis discusses our findings.

---

† Registered trademark of AT&T in the USA and other countries.

Table of Contents

Chapter	Page
Chapter One: Introduction	1
1.1. Motivations and Objectives	1
1.2. Organization of the Thesis	3
1.3. Overview of Logic Programming	3
1.3.1. Background	3
1.3.2. Syntax	4
1.3.3. Semantics	5
1.3.4. Unification	5
1.3.5. Computation Model	6
1.3.6. Prolog	8
1.3.7. Overview of A Prolog Interpreter	8
Chapter Two: Techniques for Solving CSPs	11
2.1. General Techniques	11
2.2. Techniques in Logic Programming	14
2.2.1. Intelligence Backtracking	15
2.2.2. Coroutining	16
2.2.3. CLP	18
2.3. Techniques in Artificial Intelligence	20
2.3.1. Label Inference	20
2.3.2. Value Inference	21
2.3.3. ALICE	21
2.3.4. Bernard	22
2.4. Summary	23
Chapter Three: CHIP	24
3.1. Mu-Prolog	24
3.2. Finite Domains	25
3.3. New Inference Rules	27
3.3.1. Forward Checking Inference Rule	28
3.3.2. Looking Ahead Inference Rule	31
3.3.3. Partial Looking Ahead Inference Rule	34
3.4. Built-in Predicates	36
3.4.1. Variables Instantiation	36

3.4.2. Specialization of Inference Rules 37
3.4.3. Branch And Bound 38
3.4.4. The Delay Predicate 39
3.5. Summary 39
Chapter Four: WUP3.F 41
4.1. Overview of WUP 41
4.1.1. Memory Organization 41
4.1.2. Unification Algorithm 43
4.1.3. The Interpreting Algorithm 44
4.1.4. Modules 46
4.1.5. Memory Saving Techniques 46
4.2. Finite Domains 47
4.2.1. Internal Representation 47
4.2.2. Domain Declaration 48
4.2.3. Unification For Finite Domains 49
4.3. A Model of the Delay Mechanism 49
4.3.1. Boizumault's Model 50
4.3.2. A New Model 52
4.3.2.1. The Checking Routine 52
4.3.2.2. The Expanded Unification Algorithm 55
4.3.2.3. The Activation Mechanism 56
4.4. Special Inference Rules 58
4.4.1. Forward Checking Inference Rule 58
4.4.2. Looking Ahead Inference Rule 60
4.4.3. Integration with the Interpreter 62
4.5. Built-in Predicates 65
4.5.1. Variables Instantiation 65
4.5.2. Specialization of Inference Rules 66
4.5.2.1. Specializations of FCIR 67
4.5.2.2. Specialization of LAIR 67
4.5.2.3. Specialization of PLAIR 68
4.5.3. Branch and Bound 69
4.6. Conclusion 70
Chapter Five: Performance Analysis 71

5.1. Interpretation Criteria 71
5.1.1. Execution Time 71
5.1.2. Memory Usage 72
5.1.3. Strategies Employed 72
5.1.4. The Environment 73
5.2 Examples 74
5.2.1. The N-queens Problems 74
5.2.2. A Logic Puzzle 77
5.2.3. The Mastermind Game 79
5.2.4. Map Coloring 80
5.2.5. A Scheduling Problem 82
5.3. Summary 82
Chapter Six: Conclusion 84
6.1. Summary of Results 84
6.2. Future Directions 85
6.3. New Developments in Constraint Logic Programming 85
6.3.1. CAL 85
6.3.2. CHIP 86
6.3.3. Others 87
References 89
Appendix A1 92
Appendix A2 93
Appendix A3 94
Appendix A4 96
Appendix A5 115

List of Tables

Table		Page
5.1	Relative Speeds for Different Prolog Systems	74
5.2.	Programs Written for the N-queens Problem	75
5.3.	Results of the N-queens Problem	77
5.4.	Results of the "SEND + MORE = MONEY" Puzzle	79
5.5.	Results of the Mastermind Game	79
5.6.	Results of the Map Coloring Problem	81

List of Figures

Figure	Page
1.1. An SLD-tree	7
4.1. A PC_WORD	41
4.2. Two Types of Stack Frames	42
4.3. A Linked List of Delayed Calls	53
4.4. Two Views of the Delay of p(X,Y)	54
4.5. Two Views of the Delay of p(X,Y) and q(X,Y,Z)	55
4.6. Three Types of Variables in WUP3.F	63
5.1. Results of the N-queens Problem for $5 \leq N \leq 20$	76

Chapter One
Introduction

This chapter explains the motivation and objectives of the research described in this thesis. It also describes the organization of this dissertation and provides a general overview of logic programming.

1.1. Motivation and Objectives

Constraint satisfaction problems (CSPs) are an important class of problems in Computing Science. Problems such as planning, vision, and scheduling can be easily expressed as CSPs. A CSP can be defined in the following way [vaD86]:

> *Assume the existence of a finite set J of variables $X_1, \ldots, X_n$. Suppose each variable X_i takes its values from a finite set U_i called the domain of the variable. A constraint C can be seen as a relation on a nonempty subset $I = \{Y_1, \ldots, Y_m\}$ of J which defines a set of tuples $<U_1, \ldots, U_m>$. A constraint satisfaction problem is to determine all the possible arrangements f of values to variables such that the corresponding value assignment satisfies the constraints.*

This class of problems is theoretically decidable because the domain of each variable is finite[van87]. Nevertheless, due to the complexity of these problems, a general algorithm that solves the whole class of problems will require exponential time in problem size [van87]. Although several techniques can be combined to solve a particular instance of a CSP, backtracking is by far the most commonly used technique.

Logic programming offers appealing features for solving CSPs. In this paradigm, constraints can be expressed easily in two ways: by means of predicates which enforce relationships among their parameters and by means of the unification mechan-

ism which creates (and solves) equality constraints between Herbrand terms [vaD86]. Moreover, its support of non-determinism allows natural formulation of backtracking programs: a CSP can be expressed naturally as a logic program using either the *generate and test* approach or the *standard backtracking* approach. These formulations, however, often lead to inefficient logic programs because they are oriented to recovering from failures rather than avoiding them. The search space is pruned only after failures are encountered. This approach is generally considered as the *a posteriori* approach.

Consistency techniques(CTs), on the other hand, support the *a priori* approach: constraints are used to prune the search space before failures are encountered. The main idea is to spend more time in each node of the search tree to remove combinations of values that would never appear in a solution. Thus, failures are detected earlier so that useless generations can be avoided. Moreover, the amount of backtracking and number of constraint checks is reduced [van87].

Naturally, a practical approach is to embed CTs in logic programming. CHIP represents such an effort [van87]. It supports *a priori pruning*. Furthermore, its implementation of CTs is in the form of special inference rules and a number of built-in predicates. These extra features are based on Mu-Prolog's special computation rule which is not found in most Prolog systems. If CTs are to be used widely in logic programming, it should be incorporated into conventional Prolog systems. The work in this thesis represents such an effort; we have successfully integrated CTs into the Waterloo Unix Prolog (WUP). An appropriate delay mechanism for subgoals, in which CTs can be implemented, has been identified. In addition, a performance analysis on our implementation is carried out. Findings of these aspects are discussed in the remainder of this thesis.

1.2. Organization of the Thesis

The organization of the thesis is as follows. The rest of this chapter describes the basic concepts in logic programming. In Chapter Two, constraint satisfaction techniques are examined. Some of the basic approaches are evaluated. In Chapter Three, the overall structure and the various features of CHIP are described in detail. In Chapter Four, the implementation developed by the author, called WUP3.F, is discussed. Then, the result of a performance analysis of WUP3.F is explained in Chapter Five. Finally, Chapter Six concludes this research with a summary of the findings.

1.3. Overview of Logic Programming

1.3.1. Background

Logic programming originated from advances in automatic theorem proving. It has a short history. In 1965, Robinson introduced the resolution principle [Rob65]. This principle provided a sound and complete proof procedure for first-order logic that can be efficiently implemented on a computer [Llo84]. In 1972, the first Logic Programming Language was implemented in ALGOL-W by Roussel at Marseille; this interpreter was named PROLOG (**PRO**gramming in **LOG**ic) [Llo84]. It was only after Kowalski's formulation of the procedural interpretation in 1974 did logic programming become better understood as a practical programming paradigm [Kow74]. Since then, there have been numerous implementations (e.g., [ClM80, Rob77, War77]). The Japanese Fifth Generation Computer Systems Project, which was announced in 1981, further stimulated world-wide interest in logic programming. Indeed, there are many active research areas: some of them are parallel processing of logic program, Prolog compilers, intelligent backtracking, and constraint logic programming [Hog84].

1.3.2. Syntax

Logic programming has a simple syntax in which several constructs are defined. A term is either a variable or an expression $f(t_1, \ldots, t_n)$ where f is a function symbol and t_i's are terms. 0-ary functions are constants. An atom is an atomic formula $p(t_1, \ldots, t_n)$ where p is a n-ary predicate symbol. A literal is an atom or the negation of an atom and a clause is a conjunction of literals. All variables occurring in a clause are universally quantified. A program clause is a clause of the following form:

$$A \leftarrow B_1 \,\&\, \cdots \,\&\, B_n ;^{1}$$

It contains precisely one positive literal A which is called the head. The remaining $B_1 \,\&\, \cdots \,\&\, B_n$ is called the body of the program clause. A logic program is a finite set of program clauses. A goal clause, or simply a goal, is a clause without the head; i.e., $\leftarrow B_1 \,\&\, \cdots \,\&\, B_n$ where each B_i is called a subgoal. A Horn clause is a clause which is either a program clause or a goal clause.

In Kowalski's procedural interpretation, three types of Horn clauses have specific meanings in a logic program. First, a program clause represents a procedure declaration. Its head is the procedure name and its body is the body of the procedure. Second, a unit clause is a program clause that has an empty body; i.e., $A \leftarrow$. It represents an empty procedure: a call to the procedure always returns true. Third, an empty clause, denoted by $\square$, is a clause that does not have a head nor a body. It represents a contradiction.

---

[1] For the rest of this thesis, the syntactic convention of WUP is followed: first, a variable begins with an uppercase letter; second, an identifier begins with a lowercase letter; third, the ampersand is the conjunction symbol; and fourth, the semicolon is the end of clause symbol.

1.3.3. Semantics

It is possible to assign an informal meaning to a program clause. For instance, $A \leftarrow B_1 \& \cdots \& B_n$ can be interpreted as "if $B_1 \& \cdots \& B_n$ are true, then A is true [Llo84]." Nevertheless, one of the prominent features of logic programming is its well-defined semantics. van Emden and Kowalski investigated the formal semantics of logic programming languages and described three types of semantics [vaK76]. First, operational semantics defines the denotation of a n-ary predicate p as the set of all tuples $<t_1, \ldots, t_n>$ such that $p(t_1, \ldots, t_n)$ is derivable from the set of program clauses P. Second, model-theoretic semantics defines the denotation of p as the set of all tuples $<t_1, \ldots, t_n>$ such that $p(t_1, \ldots, t_n)$ is logically implied by P. Third, fixpoint semantics defines the denotation of a recursively defined procedure to be the minimal fixpoint of a functional transformation associated with the procedure definition. That is, the denotation of p is the set of all tuples $<t_1, \ldots, t_n>$ such that $p(t_1, \ldots, t_n)$ is a member of the least Herbrand Model. These three types of semantics were shown to be equivalent by van Emden and Kowalski.

1.3.4. Unification

Unification is a uniform mechanism for parameter passing, data selection, and data construction in logic programming [Llo84]. In procedural languages, one is interested in the output of a program; whereas in logic programming, one is interested in the variable bindings computed during the execution of a logic program.

Basically, a binding is of the form v/t where v is a variable and t is a term distinct from v. A substitution θ is a finite set of bindings usually denoted as $\{v_1/t_1, \ldots, v_n/t_n\}$ where $v_1, \ldots, v_n$ are distinct. Every occurrence of the variable v_i is replaced by t_i when a substitution θ is applied to an atom G, denoted as $G\theta$. Two atoms

G and H are unifiable if there exists a unifier (i.e. a unifying substitution) ρ such that $G\rho = H\rho$. For example, consider two atoms p(X,b) and p(a,Y). If they are unified, the unifier is ρ = { X/b, Y/a } because $G\rho = H\rho$ = p(a,b). A unifier ρ is the most general unifier (mgu) if for any unifier γ, it is possible to find a substitution δ such that $\gamma = \rho\delta$. A unification algorithm is an algorithm that computes the mgu of a set of atoms [Rob65].

1.3.5. Computation Model

Computation in logic programming is based on a proof procedure called SLD-resolution [Kow74]; it is a refinement of Robinson's resolution principle [Rob65]. The procedure is similar to proof by contradiction: the negative of the formula to be proved is included as axioms with those defined in a logic program, and SLD-resolution is used to prove that the inclusion leads to a contradiction.

Specifically, the proof procedure consists of a sequence of goal reduction cycles. Each cycle involves three steps as follows:

1) Subgoal selection: a subgoal A_i is arbitrarily selected from the current goal G_i of the form $\leftarrow A_1 \,\&\, \cdots \,\&\, A_n$;

2) Procedure selection: a program clause C of the form $A \leftarrow B_1 \,\&\, \cdots \,\&\, B_m$ is chosen, and the unification algorithm computes θ as the mgu of A_i and A.

3) Goal reduction: G_i will be reduced to a new goal G_{i+1} by replacing A_i with the body of C and applying θ throughout the entire new goal. Thus, G_{i+1} becomes $\leftarrow(A_1 \,\&\, \cdots \,\&\, A_{i-1} \,\&\, B_1 \,\&\, \cdots \,\&\, B_m \,\&\, A_{i+1} \,\&\, \cdots \,\&\, A_n)\theta$. G_{i+1} is called the resolvent of C and G_i.

This cycle is repeated until either a failure occurs (when there are no more procedures to be selected) or an empty goal is reached. For example, consider the following logic program:

```
father(a,b);
father(b,c);
```

grandfather(X,Y) ← father(X,Z) & father(Z,Y);

with the goal ← grandfather(X,Y). The following steps are observed if the left-most subgoal is always chosen to be reduced:

← grandfather(X,Y);
← father(X',Z') & father(Z',Y'); { X = X', Y = Y' }
← father(b,Y"); { X' = a , Z' = b, Y' = Y"}
□ { Y" = c }

The computation can also be depicted as an SLD-tree. For example, the SLD-tree for the above execution is:

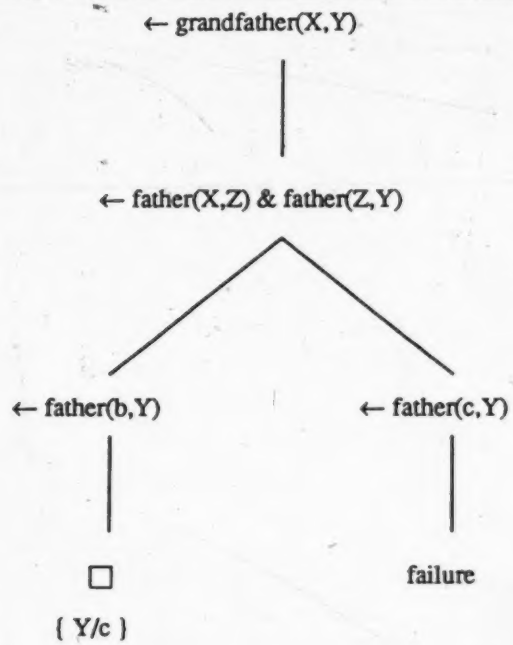


Figure1.1: An SLD-tree

In fig 1.1, each branch of the tree represents a derivation.

From the execution, two sources of non-determinism in the execution are observed: first, any subgoal in a goal can be selected; second, any procedure whose head is unifiable with the selected subgoal can be chosen. The strategy of choosing a

subgoal is called a computation rule whereas the strategy of selecting a suitable procedure for matching is called a search rule.

1.3.6. Prolog

Prolog was the first successful, and is still the most widely-used, implementation of logic programming. It uses the standard computation rule which always selects the left-most subgoal from a goal and the standard depth-first search rule which tries procedure declarations in their textual order. This scheme allows Prolog to be implemented efficiently in the von Neumann architecture of a computer because a stack can be used to implement the depth-first search. Basically, each record on the stack represents the current resolvent. An instance of the stack represents the branch of the search tree that is currently being investigated. Whenever there is a match between the current selected subgoal and the head of a program clause, the new resolvent is pushed onto the stack. If, however, there is no such match, the top of the stack is popped. The matching, then, continues with the selected subgoal of the current top of the stack. Thus, the computation essentially becomes an interleaved sequence of pushes and pops on the stack [Llo84]. Although this stack implementation allows efficient execution, it introduces incompleteness into the system. For instance, solutions to the right of an infinite branch of the search tree may never be found [Llo84]:

1.3.7. Overview of A Prolog Interpreter

A conventional Prolog interpreter accesses two important data areas in memory. The first area is the input heap (or the pure code area). It contains the compacted and codified form of an input logic program. Before program execution, all clauses of the program are loaded into the input heap. They are then linked together and arranged by their names and arities, yet the textual order of the original program is preserved. With

clauses arranged in this fashion, the interpreter can locate a candidate clause quickly during unification.

The second area is the runtime environment. Its structure depends largely on the strategy used to represent a structured term that contains variables. In fact, there are two common strategies [Bru82]. The first strategy is called structure sharing which allows pure code to be shared among instances of a term. A term is represented by two pointers: the skeleton pointer and the environment pointer. The former points to a skeleton which defines the "shape" for the term and the latter points to a variable environment in which the values of the variables in the term can be accessed. Using this scheme, the interpreter can construct new terms with little cost. Yet accessing to the content of a term is slow because much dereferencing is required. The second strategy is called structure copying [Mel82]. According to this strategy, a new copy of the term has to be created when a variable is bound to a term that contains variables. Thus, term construction is slow while accessing a term is fast because less dereferencing is required. Since our investigation is on WUP, the following description is based on this structure copying interpreter.

There are three stacks in the runtime environment of WUP. The first stack is the trail stack. It records the addresses of the variables that are bound during unification. Upon backtracking, these variables are reset to "undefined". The second stack is the copy (or global) stack. It serves as a heap for structures that are created during program execution. Every time that a variable is bound to a term containing variable, a new copy of the term is created on the copy stack. The third stack is the runtime (local) stack. A frame is created on this stack after a candidate clause has been selected to be unified with the current goal. Each frame is divided into two areas: environment and control areas. The environment area keeps track of variable bindings. It contains a

number of memory slots; one for each unique variable in the current program clause. The control area keeps track of the flow of control. It contains several pointers and the number of pointers depends on whether the current computation is a deterministic one or a non-deterministic one. A computation is deterministic if there is only one candidate clause for the current subgoal. Otherwise, the computation is non-deterministic. A deterministic computation requires the following pointers:

FATHER — a pointer to the parent goal of the current subgoal. It links up all the ancestors of the current subgoal; it maintains the structure of the proof tree;

RETURN — a pointer to the next subgoal which is to be solved upon successful completion of the current one.

A non-deterministic computation, on the other hand, requires several additional pointers as follows:

NEXT CLAUSE — a pointer to the next candidate clause for the current subgoal. Upon backtracking, the pointer is used to locate this clause.

BACK — a pointer to the most recent backtracking point. It helps to locate the last backtracking point. Upon backtracking, the segment of the run-time stack above this point is discarded.

RESET — a pointer to the trail that records the top of the trail at the time of creation of the current frame. Upon backtracking, the segment of trail above[2] this point is discarded. Variable bindings recorded in this portion are reset to "undefined".

COPY — a pointer to the copy stack that records the top of the copy stack at the time of creation of the current frame. Upon backtracking, the segment of copy stack above this point is discarded.

Most Prolog implementations allow built-in predicates to be defined. These predicates serve two functions. First, they allow the use of non-logical constructs such as cut and negation as failure [Llo84]. Second, they help to speed up execution of the interpreter in some domains such as arithmetic and strings manipulation.

---

[2] In this thesis, it is assumed that all stacks grow upwards.

Chapter Two
Techniques for Solving CSPs

This chapter describes some of the techniques for solving CSPs. It first discusses the general technique and some of its variants; backtracking is the basis of this technique. Then, similar techniques in logic programming are examined. Lastly, a few techniques developed in the field of Artificial Intelligence are explained.

2.1. General Technique

Backtracking is the most often used method in solving CSPs. Basically, backtracking can solve a problem that can be transformed into the assignment of values to a tuple of variables of the form $(v_1, \ldots, v_n)$ so that a certain property, say $P(v_1, \ldots, v_n)$, is satisfied. Each of the variables v_i has a finite domain D_i of constants. Backtracking will instantiate the variables sequentially, and each partial solution $(e_1, \ldots, e_k)$ is checked to see if it satisfies a criterion function P_k. If $P_k(e_1, \ldots, e_k)$ does not hold, a new value from D_k is selected to instantiate v_k. If all the values from D_k have been used, the system backtracks, and a new value will be given to v_{k-1}. Otherwise, if $P_k(e_1, \ldots, e_k)$ holds, the system will instantiate the next variable, i.e. v_{k+1}. The main advantage of backtracking is that it is possible to explore the entire search without enumerating all the tuples even though in the worst case, all tuples have to be checked. For instance, a partial assignment of variables may falsify a criterion function and backtracking will generate a new partial assignment instead of expanding the partial assignment.

As the criterion function P_k is not defined in the general backtracking procedure, it is possible to have variants for the procedure. In fact, four variants of backtracking have been proposed to solve CSPs: generate and test, standard backtracking, forward checking, and looking ahead [van87]. Each approach has a different criterion function. Basically, generate and test requires that the variables $v_1, \ldots, v_k$ are instantiated; that is, a complete assignment is made. Standard backtracking, in addition to a complete assignment, requires that the newly instantiated variable must be consistent with all other instantiated variables. For example, if the problem is to find a sorted list of elements among the domains, $P_k(e_1, \ldots, e_k)$ will require that $e_1 < \cdots < e_k$. Similarly, forward checking requires that each variable which has not yet been assigned has at least one consistent value with other assigned variables. Lastly, for looking ahead, each variable that has not yet been assigned must have at least one consistent value with other variables that have not been assigned.

Informally, the relative efficiency of the four approaches can be compared by studying their criterion functions. First, in the generate and test approach, every possible tuple is tested because the search space is not pruned. As the size of the program grows, this approach becomes extremely inefficient. Second, in the standard backtracking approach, pruning of the search space occurs only after a failure is detected, a kind of *a posteriori pruning*. Even though the approach already represents a significant improvement over the generate and test approach (i.e. pruning is performed passively), it suffers from the thrashing problem which has the following symptoms [Mac77] :

1) repeat useless generation: this situation occurs when there is a certain value in the domain of a variable which always falsifies the criterion function. For instance, if $v_k = c$ always falsifies the criterion P_k, a new instantiation of any of the variables $v_1, \ldots, v_{k-1}$ will eventually make the procedure reinstantiate v_k to c and cause a failure again.

2) bad backtracking point: this situation occurs when the instantiation of v_i causes the failure of P_j where $j \geq i$. The system should backtrack to give a new value

for v_i instead of assigning different values for the variables $v_{i+1}, \ldots, v_j$.

3) late detection of failure: this situation occurs when the values of two variables v_i and v_j lead to the failure of P_k where $i < j < k$. The system will needlessly assign different values to the variables $v_{j+1}, \ldots, v_k$.

With these drawbacks, the standard backtracking approach is not efficient for solving CSPs. Consequently, the forward checking approach and the looking ahead approach are proposed to avoid the inefficiency which plagues the first two approaches. The main idea behind these lookahead approaches is that more time is spent on the effect of instantiating a new variable. Inconsistent values are removed from domains of the variables that have not yet been assigned so that failures are detected as soon as possible. Thus, each instantiated variable will not cause any unnecessary failure, especially those related to the three symptoms of thrashing.

However, it is difficult to compare the efficiencies of different approaches without actually writing and running the programs [Knu75]. van Hentenryck, therefore, used a statistical model to compare the standard backtracking approach with the forward checking approach [van87]. The purpose of the comparison is to "give a mathematical explanation of the greater efficiency of lookahead schemes over standard backtracking and to define the conditions under which this explanation holds" [van87]. The number of constraint checks is used as a measure of the time complexity of the algorithms. Without going into the full detail of this analysis, the findings are summarized as follows:

1) the standard backtracking approach is more efficient than the forward checking approach only when the probability of success of a constraint is small (i.e. < 30%); in all other cases, the latter is more efficient.

2) spending more time in the beginning of the tree search pays off rather quickly and achieves an overall improvement.

3) the forward checking approach is always more efficient than the standard backtracking approach for difficult problems, such as those that require a lot of

constraint checks.

The findings of the analysis support the proposition that lookahead approaches are more efficient.

2.2. Techniques in Logic Programming

Logic programming has much to offer in constraint solving. First, unification solves equalities among terms. Second, the lack of orientation of parameters provides a mechanism whose expressive power is similar to that of equations. Third, the non-determinism of the search procedure accommodates the generations of labelling, i.e. instantiation of variables [Gal87]. Fourth, simple constraints can easily be implemented as built-in predicates, e.g. the inequality constraint $X \neq Y$. Lastly, composite constraints can be expressed as logic programs [van87]. For instance, in a problem that requires all the variables to have distinct values, the all_different($[X_1, \ldots, X_n]$) predicate can be implemented as a logic program which makes sure that $X_i \neq X_j$ for all i and j, $i \neq j$.

The most commonly used approach to express a CSP as a logic program is the generate and test approach. Basically, a program written in this approach can be divided into two modules: the generate module and the test module. The generate module computes the value assignment for all the variables, and the test module checks if the assignment satisfies the given set of constraints. However, the generate and test approach is not efficient because it does not prune the search space (see section 2.1). Even though more efficient approaches, such as standard backtracking, forward checking, and looking ahead can be used, the resulting program, on one hand, requires more programming effort and, on the other hand, becomes more difficult to understand and to maintain [van87]. Furthermore, the resulting program still uses Prolog's chronological

backtracking which suffers from all three symptoms of the thrashing problem.

Nevertheless, much research has been devoted to the improvement of the efficiency of Prolog. Some deal with the overall improvement of program execution such as compilation of Prolog programs [War77] and parallel processing. Others deal specifically with Prolog's control strategies. In the following, two pieces of research that belong to the latter category are reviewed: intelligent backtracking [Won88] and coroutining [ClM80]. Moreover, Constraint Logic Programming [HJM87], which is the aim of this research, becomes an increasingly active area. The work in CLP(R) is compared with the approach of embedding CTs in logic programming.

2.2.1. Intelligent Backtracking

Intelligent backtracking is a refinement of the chronological backtracking of Prolog that aims to alleviate the problem of thrashing. In chronological backtracking, the system will backtrack to the most recent backtracking point. One of the problems with chronological backtracking is related to bad backtracking points. Consider the following program and goal:

```
p(a,b);
p(b,c);
q(a);
q(b);
r(c);
← p(X,Y) & q(Z) & r(Y).
```

After the first two subgoals have been proved, the system obtains the substitution {X/a, Y/b, Z/a} and tries to prove r(b). It fails. The system then backtracks to q(Z) and reproves q(Z) to get a new binding for Z, i.e. { Z/b }. After this, it proves r(b) and, obviously, it fails once again. Clearly, the failure of r(Y) has nothing to do with the binding of Z. Instead, the system should backtrack to p(X,Y) so that a new binding for Y can be obtained. The system, however, backtracks needlessly to obtain all the new

bindings for Z. In sum, q(Z) is a bad backtracking point.

In intelligent backtracking, the source of failures is analyzed. When the system backtracks, it always backtracks to a productive choice point. There are many methods to locate these choice points such as static data dependency analysis, generator-consumer analysis, and deduction analysis [Won88]. Unfortunately, the problem of finding the best possible backtrack points is intractable; that is, the complexity of the algorithm for finding these points is exponential[1] [Wol86]. Usually, heuristics are used instead. The overhead involved in finding backtracking points may slow down program execution.

Even though intelligent backtracking is able to give Prolog a better performance, it does not eliminate the problem of thrashing [van87]. The search space is still pruned only after a failure is encountered. Thus, the overall strategy is still the *a posteriori* approach. Lee Naish commented about this approach: "It is better to avoid failures than to react to them intelligently [Nai85]."

2.2.2. Coroutining

Another improvement on Prolog's control strategy is the coroutining mechanism. It enables sophisticated computation rules to be defined. In coroutining, a generate and test program can be written in a way such that the test module is placed before the generate module. If there is insufficient information for a test to be carried out, it is delayed. As soon as sufficient information is available, the test is reactivated and executed. In other words, the test can be used as soon as possible without waiting for the end of the generation of values [DSv87]. Thus, standard backtracking can be realized in

[1] According to [Wol86], the problem of finding all maximal unifiable subsets to those of certain sizes, for use with heuristics, are shown to be NP-hard.

the generate and test approach.

There are many approaches for adding the coroutining mechanism to Prolog. The basic idea is to enhance the computation rule so that more processing is carried out to decide whether a selected subgoal should be executed or delayed. The following two examples illustrate this point. First, in PrologII [Col82], an instance of the freeze predicate, say freeze(X,p(X,Y,Z)), contains the control information that X must be instantiated before p(X,Y,Z) can be executed. Second, in Mu-Prolog, a wait declaration, say wait p(+,+,-), means that the first two arguments must be constructed before p can be executed [Nai85]. Usually, different criteria for execution are used for different implementations. Coroutining, therefore, allows the natural formulation of generate and test to be implemented with the efficiency of standard backtracking.

It is not difficult to see that constraints are still used passively; that is, the search space cannot be pruned until failures are encountered. These failures are due to the instantiation of inconsistent values of variables. Thus, coroutining does not produce any improvement beyond the standard backtracking approach [van87]. In addition, it does not handle the interaction among constraints [DSv87]. For instance, assume that the coroutining mechanism is used in two built-in predicates, $\geq$ and $\leq$, so that these two predicates can be executed only if all the parameters are ground. Consider the following program and the two goals G1 and G2:

$p(X,Y) \leftarrow X \geq 5 \;\&\; Y \leq 5;$
$q(X,Y) \leftarrow X \geq 8 \;\&\; Y \leq 2;$
G1: $\leftarrow p(Z,Z);$
G2: $\leftarrow q(Z,Z);$.

During the execution of G1, the system suspends the resolvent $Z \geq 5 \;\&\; Z \leq 5$; but, obviously, Z should get the value 5. On the other hand, during the execution of G2, the system suspends the resolvent $Z \geq 8 \;\&\; Z \leq 2$; but, the resolvent contains an inconsistency.

Indeed, coroutining alone is not sufficient for constraint solving.

2.2.3. CLP

Among the many extensions of Prolog, CLP is the closest to the approach of embedding CTs in logic programming where unification in CLP is replaced by the concept of constraint solving. CLP has its origin in the Logic Programming Language Scheme(or simply the Scheme) which represents an effort to preserve the unique semantic properties of logic programming among its extensions. Basically, the syntax of the Scheme is the syntax of Definite Clause. Its domain of computation is left unspecified but it is assumed to be definable by a unification complete equality theory; that is, an equality theory E dictates that equality holds only if E-unification is possible [JLM86]. Furthermore, its interpreter is based on SLD resolution and an appropriate, generalized unification algorithm [JaL87]. One of the advantages of the Scheme is that the programmer who uses an instance of the Scheme can work directly in the intended domain of discourse while the semantics of the instance is being taken care of [van87]. Moreover, efficiency is greatly improved by the use of constraint solving techniques over specific domains.

CLP is an extension of the Scheme. Its interpreter is based on SLD resolution and an appropriate constraint-solver which can handle constraints in the given computation domains. There are three characteristics for the constraint-solver [van87]: first, it is a decision procedure for the class of constraints in the given computation domain — it can always decide if there is a solution for a set of constraints or not; second, it is a blackbox to the user — the user has no control and knowledge about it; and third, it has to be incremental.

CLP(R) [HJM87, JaL87] is an instance of CLP. Its computation domain is real numbers; and its constraint-solver can handle constraints in the form of linear equations and inequations. In CLP(R), equations are handled by the Gaussian Elimination method whereas inequalities are solved by a modified Simplex method [Las87]. Yet non-linear constraints are delayed until they become linear.

The approach of CLP is quite similar to the approach of embedding CTs in logic programming (the CTs approach) because both of them are based on the idea of using constraints in an *a priori* way. However, there are some fundamental differences between the two approaches, and van Hentenryck has the following arguments [van87]. Firstly, the CTs approach attempts to solve discrete combinatorial problems which require a combination of backtracking and constraint manipulation techniques. Each problem in this class requires specific handling. Yet CLP is not designed to solve a specific class of programs. Secondly, the CTs approach does not require a complete constraint-solver because discrete combinatorial problems are best regarded as searches. A complete problem solver for these problems is inefficient, e.g. [Fre78]. Thirdly, the CTs approach enables a programmer to use any strategies and representations he likes to exploit the specific nature of his problem. This freedom is not available in CLP because the constraint solver is a black box to the users. Lastly, the CTs approach provides general mechanisms such as forward checking and looking ahead for those constraints that can be expressed as logic programs. In sum, it should be reasonable to say that CTs can provide better support for discrete combinatorial problems.

Besides CLP(R), Prolog III [Col87] (an improved extension of Prolog II [Col82]) can also be viewed as an instance of the scheme even though it is not presented in this way.[2] Its computation domain is rational numbers. The constraint-

[2] In [JLM86], Prolog II is shown to be an instance of the Scheme.

solver of Prolog III has two submodules: one for processing Boolean Algebra and the other for processing inequalities. The latter module is based on a modified simplex algorithm.

2.3. Techniques in Artificial Intelligence

In many AI systems, constraints are used to reason about quantities. There are many techniques for constraint solving. In this section, two general techniques are explained and the problem solver ALICE is also mentioned. Lastly, a programming language that can solve a class of CSPs is reviewed.

2.3.1. Label Inference

In label inference, each variable is associated with a set of possible values, and constraints are used to restrict these values. Suppose that there is a constraint applied to a particular variable. It requires that the variable must be even. In label inference, all the odd values in set of possible values of the variable will be eliminated.

The first significant work was introduced by Mackworth [Mac77] in which he proposed three notions — node, arc, and path — of consistency and devised algorithms to enforce them. In fact, these notions are remedies for the three causes of the thrashing problem described in section 2.1. The main idea is to eliminate local inconsistencies before any real processing is carried out to find the complete solution. Even though these algorithms cannot find the solution, they can reduce the search space. Thus, they are best viewed as preprocessing techniques.

Expanding on Mackworth's work, Freuder proposed the notion of k-consistency and developed an algorithm to synthesize a set of constraints [Fre78]. The algorithm utilizes both local and global propagation; however, it is inefficient for solving

problems that have larger number of constraints.

2.3.2. Value Inference

In value inference, constraints are used to assign values for variables from those that are already assigned. Each constraint is usually defined as a set of rules. For instance, given a constraint that represents an *AND* gate with inputs X,Y, and output Z, the following set of rules defines the constraint [van87]:

- if Z = 1, then X = Y = 1;
- if X = 0 or Y = 0, then Z = 0;
- if X = 1 and Y = 1, then Z = 1;
- if Z = 0 and X = 1, then Y = 0;
- if Z = 0 and Y = 1, then X = 0;
- if X = 1, then Z = Y;
- if Y = 1, then Z = X;

Notice that some of the rules are redundant and each rule identifies a particular use of the constraint. As soon as the condition of a rule is satisfied, the rule can be applied. Such an application may instantiate other variables and will lead to possible applications of other rules. This cycle of instantiation of variables and application of rules introduces a data-driven computation.

It is not difficult to realize that the search space is reduced in an *a priori* way because values are propagated as soon as they are generated. However, value inference is only applicable for constraints which can be expressed as equations [van87]. This restriction makes the technique insufficient for solving combinatorial problems.

2.3.3. ALICE

ALICE [Lau78] represents a class of problem solvers which relies on a general search procedure that utilizes domain specific knowledge. In each of these solvers, a specification language is used to formulate problems. Using the given specification as

input, the system will search for a solution using the *a priori* approach. For instance, ALICE has a mathematical language that incorporates knowledge in algebra, set theory, first order logic, and graph theory. It uses a depth-first search procedure which is equipped with a sophisticated constraints manipulation mechanism to find the solution. The search is also assisted by a large set of general heuristics. The outcome of the search is a function from one finite set to another which satisfies the given set of constraints [Lau78].

A major advantage of such a system is that the user is relieved from writing an algorithm to solve a particular problem. He only needs to write a specification for the problem. However, to take full advantage of the system, he has to learn every detail of the specification language. Yet these solvers have two disadvantages. First, the system is a black-box [van87] to the user. He has no control in the representation of his problem and the strategy that solves it. Second, the application range of the system is restricted by both the specification language as well as the amount of domain specific knowledge inside the system.

2.3.4. Bernard

Bernard [Lel88] is a general-purpose specification language. It uses an extended form of term writing called augmented term rewriting which has the ability to bind values to variables and to define abstract data types. These extensions make the language expressive enough to be used for general constraint satisfaction [Lel88]. Unfortunately, Bernard is primarily used for systems that handle numeric constraints; that is, its computation domain is real numbers.

In Bernard, constraints are expressed as rewriting rules, and consistency techniques can be implemented as libraries of rules. Moreover, a user can specify a specific

constraint-satisfaction system by composing a set of rewriting rules. The major strengths of Bernard are: first, it is expressive enough to handle systems of nonlinear equations; second, it is extensible; and third, it inherits computational completeness from term rewriting. However, when used as a constraint-satisfaction problem solver, Bernard has two crippling weaknesses: first, it cannot give multiple solutions since the term rewriting system is not capable of dealing with multiple values for a single variable. ; second, it cannot handle inequalities.

2.4. Summary

In this chapter, several techniques for constraint solving are discussed. In particular, four approaches of backtracking have been described. One can argue that the lookahead approaches provide better pruning than both the generate and test approach and the standard backtracking approach do. On the other hand, even with intelligent backtracking and coroutining, Prolog is unable to allow efficient formulations of CSPs. Furthermore, techniques developed in AI are shown to be insufficient for solving CSPs. Therefore, a possible approach is to embed CTs in logic programming. CHIP, which is the topic of the next chapter, is the first attempt along this line.

Chapter Three
CHIP

This chapter discusses the implementation of the CHIP language [1] in detail. It contains five sections. Section one describes Mu-Prolog which is the backbone of CHIP's implementation. Section two examines the concept of finite domains which allows CTs to be integrated into logic programming. Section three discusses the three inference rules in CHIP. Section four shows the built-in predicates that constitute the CTs in CHIP. Finally, some concluding remarks are given in section five.

Before going on to discuss the CHIP implementation, it is important to point out its three design principles:

- minimize overhead: extra overhead should not be incurred for those programs that do not use CTs;
- avoid intermediate variables: these variables cause many problems in a structure sharing interpreter;
- achieve data-driven selection of subgoals: as soon as there is enough information to execute a subgoal, it should be selected immediately.

These principles are followed in CHIP's implementation. In the following discussion, the areas where CHIP observes these rules are indicated.

3.1. Mu-Prolog

Mu-Prolog is a Prolog interpreter developed by Lee Naish at the University of

---

[1] The discussion here is based on [van87]. Besides constraints in finite domain, the CHIP language can handle constraints in Boolean Algebra and rational numbers. [DvS88] gives a good summary of the current status of CHIP. Our research concentrates on finite domains only.

Melbourne, Australia. It is used to test the feasibility of the wait declaration and its automatic generation. The wait declaration is actually a sophisticated control mechanism that allows preconditions to be defined for selection of a subgoal [Nai85].

Mu-Prolog is a simple interpreter. It has a straightforward memory organization: it uses only one stack and does not implement any memory saving techniques. Yet the novelty of Mu-Prolog is found in its computation rule. In Mu-Prolog, a subgoal can be delayed and invoked later and the scenario can be depicted as follows. After a subgoal is selected, a test is carried out to determine whether the subgoal satisfies its preconditions or not. If it does, the execution continues in the normal way. Otherwise, the subgoal is delayed. This means that the subgoal is removed from the current resolvent and a link is created between the subgoal and each of the variables that falsifies the preconditions. Later on, if one of these variables get bound, the delayed subgoal will be checked again to determine if it satisfies its preconditions. If it does, the delayed call is put back into the resolvent and is ready to be selected again.

In Mu-Prolog, the computation rule is slightly different in that the test for preconditions is carried out during unification. Moreover, whenever a subgoal is invoked, it is immediately placed back to the resolvent. This scheme will cause quite a few activations and delays of the same subgoal. It is, however, difficult to improve on this matter given the generality of the scheme [Car88, van87].

3.2. Finite Domains

The mechanism of finite domains was introduced into logic programming in [vaD86]. It allows the active use of constraints and the implementation of consistency techniques. The main idea is to associate a variable with a set of possible values. The variable can take up only one element from the set as its value. Hence, the

interpretation of a logic program is more confined. A variable that has a finite domain is called a d-variable (or simply d-var) whereas one that does not is called an h-variable (or simply h-var).

In CHIP, the domain for a d-var is characterized by three entities. The first entity is the cardinality of the domain. The second entity is a boolean array that indicates whether a particular element of the domain is removed. The third entity is a mapping between the elements of the boolean array and the elements of the domain. Generally speaking, the mapping can be any quick access method such as a binary search or a hash-table. But for specialized domains, the mapping function can be more specific. For instance, for a domain of consecutive integers, the mapping can be as follows:

$$B = V - L + 1$$

where B is the index for the boolean array, V is the value of an element, and L is the lower bound. Consider V = 9 and L = 5. The 5th (i.e. 5 = 9 - 5 + 1) entry in the boolean array tells whether the integer 9 is removed from the set or not.

In actual implementation, a domain is represented as a record. The memory used for the record is allocated from the local stack. Upon backtracking, the memory can be reclaimed. This strategy helps to minimize memory usage and, thus, the overhead.

The domain declaration of the form

domain p($d_1, \ldots, d_n$)

is used to introduce finite domains in a logic program. d_i is a domain specification which can be a constant 'h', a set of constants $\{a_1, \ldots, a_n\}$, or an expression l..u. The first specification means that the ith argument of predicate p ranges over the Herbrand universe. The second one means that the ith argument is a set of natural numbers or

character strings. The third one means that the argument is a set of consecutive natural number with lower bound l and upper bound u.

Since d-variables are treated as a distinct type of objects, the unification algorithm has to be modified to allow unification between a d-var and a term. Three possible cases are observed:

- the term is a simple variable: it is bound to the d-var;
- the term is ground (i.e. it contains no variable): if it is a member of the domain of the d-var, the d-var will be bound to the term; otherwise, the unification fails;
- the term is a d-var: the intersection of the domains of the two d-vars is computed. If the intersection is not empty, a new d-var with domain as the intersection is created. The two d-vars become pointers to the new d-var; otherwise, unification fails.

It is not difficult to see that equalities among d-variables can be solved in the third case.

3.3. New Inference Rules

Three inference rules are introduced into CHIP. They are forward checking inference rule (FCIR), looking ahead inference rule (LAIR), and partial looking ahead inference rule (PLAIR). These rules provide a formal basis for their own control mechanisms and the implementation of some specialized predicates. From a practical point of view, they are best viewed as general methods [van87]. They allow a user to specify not only which inference rule is used for a given constraint, but also what kind of preconditions must be satisfied before the constraint can be used. These rules are used when there is insufficient knowledge concerning a particular constraint. If there is sufficient knowledge, specific methods can be applied (see next section). In most applications, these rules are more effective than resolution in terms of pruning of the search space.

These inference rules can only be used in constraints which are defined as follows:

> **Definition 3.1:** *Let p be an n-ary predicate. p is a constraint iff for any ground terms $t_1, \ldots, t_n$, either $p(t_1, \ldots, t_n)$ has either a successful refutation or $p(t_1, \ldots, t_n)$ has finite failed derivations* [van87].

Yet there is no mechanism to guarantee that an atom which uses an inference rule is a constraint. This remains as a user's responsibility.

3.3.1. Forward Checking Inference Rule

Forward Checking Inference Rule (FCIR) introduces the forward checking strategy of solving constraints into logic programming. The idea is that a constraint can be solved when there is only one uninstantiated variable left. Actually, if the constraint is solved using this rule, only consistent values will remain in the domain of the variable. To apply FCIR, a constraint must be forward-checkable. The definition of forward-checkable is given as follows:

> **Definition 3.2:** *An atom $p(t_1, \ldots, t_n)$ is forward-checkable iff p is a constraint and there exists one and only one t_i which is a d-variable, with all the others being ground. This last variable ($t sub i$) is called the forward-variable* [van87].

FCIR [2] is defined as follows [van87] :

> **Definition 3.3:** *Let P be a program and $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ be a goal. G_{i+1} is derived by the FCIR from G_i and P using the substitution θ_{i+1} if the following conditions hold*
>
> [1] *A_m is forward-checkable and x^d is the forward-variable;*
>
> [2] *$e = \{a \in d \mid P \models A_m\{x^d/a\}\} \neq \emptyset$.*
>
> [3] *θ_{i+1} is*
>
> [3a] *$\{x^d/c\}$ if $e = \{c\}$;*

---

[2] The soundness and completeness results are shown in [van87].

[3b] $\theta = \{x^d / z^e\}$ *where* z^e *is a new d-variable otherwise.*

[4] G_{i+1} *is the goal* $\leftarrow(A_1 \,\&\, \cdots \,\&\, A_{m-1} \,\&\, A_{m+1} \,\&\, \cdots \,\&\, A_k)\theta_{i+1}$.

There are three observations from the above definition of FCIR. First, the inference rule solves the constraint completely. According to [4], A_m is removed from the resolvent. Second, this rule achieves *a priori* pruning. In [2], e is always a subset of d because all the inconsistent values in d are removed. Third, if there is only one value left in the domain of the forward-variable ([3a]), the forward-variable will be bound to that value.

To use FCIR, a constraint must be associated with a forward declaration which is defined as follows:

> **Definition 3.4:** *Given an n-ary predicate p, a forward declaration, unique for this predicate, is an expression of the following form*
>
> *forward* $p(a_1, \ldots, a_n)$
>
> *where the* a_i*'s are either the character 'g' or the character 'd'. p is a constraint and all the atoms having p as predicate are said to be submitted to this forward declaration* [van87].

Forward declarations are separated from the logic part of the program and do not influence the declarative semantics. They provide a general method for using forward checking in logic programming.

The semantics of the rule is encapsulated in the definition of a forward-consistent proof-procedure. The following definitions are given [van87].

> **Definition 3.5:** *An atom P submitted to a forward declaration is forward-available iff all the arguments of P corresponding to a g in the forward declaration are ground, and P is forward-checkable.*

> **Definition 3.6:** *A computation rule R is forward-consistent iff an atom submitted to a forward declaration is selected by R only when it is either ground or forward-available.*

Definition 3.7: *A proof-procedure is forward-consistent iff*

- *it uses a forward-consistent computation rule R;*
- *when a forward-available atom P is selected by R, the FCIR is used to solve P;*
- *when an atom P that is not forward-available is selected by R, normal derivation is used.*

The forward consistent computation rule is provided by the system so that the design of program can be simplified [van87]. For instance, constraints can be stated before the generators and will be selected only when they are forward-available or ground.

To make the proof procedure more efficient, the concept of forward-efficient computation rule is introduced as follows:

> **Definition 3.8:** *A computation rule R is forward-efficient iff R is forward-consistent and each time one or several atoms submitted to a forward declaration are ground or forward-available, one of them is selected by R* [van87].

A forward-efficient computation rule enables the pruning of the search space as soon as possible and introduces a data-directed computation [van87]. Since FCIR can instantiate the forward variable, the binding can cause other predicates to be forward-available. These predicates can then be used to further reduce the search space. This may lead to further instantiation of variables.

Nevertheless, the use of forward-consistent and forward-efficient rules introduces incompleteness into the language [van87]. It is possible that the computation terminates with a non-empty goal while some of the subgoals may not be selected. For instance, an inequality constraint that uses the FCIR may never be executed if both of its arguments are not instantiated during the entire execution. The system in this situation is said to be floundering [Nai85]. The user is therefore responsible for the provision of generators of values for the variables; these generators make sure that any constraint will be selected by computation rule sooner or later. This approach not only

simplifies the implementation but also allows the user to use specific knowledge about the problem to define the generators.

FCIR is implemented as recursive calls to the interpreter. Basically, whenever a predicate submitted to a forward declaration is selected, a test is carried out to determine if it is forward-available (which is the precondition for selection of a constraint submitted of forward declaration). If it is not forward-available, the constraint is delayed. Otherwise, the current state of the interpreter is saved to provide a clean environment for recursive calls to the interpreter. In each call, an element of the domain of the forward-variable in the constraint is used to bind the forward-variable. Then the ground constraint is proved. If it can be solved, the picked element is consistent and will not be removed from the domain. Otherwise, the element is removed. After all the elements are tested, there are three possible outcomes. First, several values are consistent. They form the new domain for the forward-variable. Second, there is only one value left. The forward-variable will be bound to that value. Third, there is no value left. This implies that the constraint cannot be solved.

It is not difficult to see that the number of recursive calls is equal to the cardinality of the domain of the forward-variable. For each call, it is possible to eliminate a value from the domain. In this sense, FCIR is a useful inference rule.

3.3.2. Looking Ahead Inference Rule

Looking Ahead Inference Rule (LAIR) is used when more than one variable is left uninstantiated. It reduces the set of possible values of these variables. The definition of lookahead-checkable is given below:

Definition 3.9: *An atom* $p(t_1, \ldots, t_n)$ *is lookahead-checkable iff*

- *p is a constant;*
- *there exists at least one t_i which is a d-variable and each of the other arguments is either ground or a d-variable. The d-variables in $t_1, \ldots, t_n$ are called the lookahead-variables* [van87].

LAIR[3] is defined as follows [van87] :

Definition 3.10: *Let P be a program, $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ be a goal. G_{i+1} is derived by the LAIR from G_i and P using the substitution θ_{i+1} if the following conditions hold:*

[1] *A_m is lookahead-checkable and $x_1, \ldots, x_n$ are the lookahead-variables of A_m which range respectively on $d_1, \ldots, d_n$;*

[2] *for each x_j, $e_j = \{ v_j \in d_j \mid$ there exist $v_1 \in d_1, \ldots, v_{j-1} \in d_{j-1}$, $v_{j+1} \in d_{j+1}, \ldots, v_n \in d_n$ such that $A\theta$ is a logical consequence of P with $\theta = \{ x_1/t_1, \ldots, x_n/t_n \}\} \neq \emptyset$;*

[3] *z_j is the constant c if $e_j = \{c\}$ or a new variable ranging over e_j otherwise;*

[4] $\theta_{i+1} = \{x_1/z_1, \ldots, x_n/z_n\}$;

[5] *G_{i+1} is either*

[5a] $\leftarrow(A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta_{i+1}$ if at most one z_i is a d-variable or

[5b] $\leftarrow(A_1, \ldots, A_k)\theta_{i+1}$ otherwise;

LAIR is a generalization of FCIR [van87]. When there is only one lookahead-variable, as in [5a], it behaves like FCIR. Otherwise, it cannot solve the constraint in most cases, as in [5b]. LAIR can be seen as a general mechanism for enforcing k-consistency between lookahead variables.

A lookahead declaration is used to introduce LAIR into logic programming and is defined as:

Definition 3.11: *Given a n-ary predicate P, a lookahead declaration, which is unique for this predicate, is an expression of the following form:*

---

[3] In [van87], the soundness of LAIR is proven. However, there is no result for completeness as it is generally not possible for a k-consistency algorithm to solve most of the problems.

lookahead $p(a_1, \ldots, a_n)$

where a_i*'s are either 'g' or 'd'. p is a constraint and all the atoms having p as predicate are said to be submitted to this lookahead declaration* [van87].

The semantics of the rule can be captured by a lookahead-consistent computation rule, given by the following definitions:

Definition 3.12: *An atom P submitted to a lookahead declaration is lookahead-available iff*

- *all the arguments of P corresponding to a g in the lookahead declaration are ground;*
- *P is lookahead-checkable* [van87].

Definition 3.13: *A computation rule R is lookahead-consistent iff an atom submitted to a lookahead declaration is selected by R only when it is either ground or lookahead-available* [van87].

Definition 3.14: *A proof-procedure is lookahead-consistent iff*

- *it uses a lookahead-consistent computation rule R;*
- *when a lookahead-available atom P is selected by R, the LAIR is used to solve P;*
- *when an atom P that is not lookahead-available is selected by R, normal derivation is used* [van87].

An efficient computation rule is even more important for lookahead declarations than for forward declarations because an arbitrary selection of atoms can induce a lot of redundancy. As a result, the atoms that are submitted to a lookahead declaration are grouped into two sets [van87]. Set Y is a set of all atoms which, if selected, will bring about new information like instantiations of variables. Set N is the set of all other atoms. Therefore, a goal can now be denoted as ←<Y,N> where Y is called the Y-part of the goal and N the N-part. Using this notation, a lookahead-cheap computation rule is defined as follows:

Definition 3.15: *A computation rule R is lookahead-cheap iff R is lookahead-consistent and if R selects only atoms in the Y-part of the goal. A computation rule R is lookahead-efficient iff R is lookahead-cheap; and if each time one or*

several atoms submitted to a lookahead declaration in the Y-part are lookahead-available or ground, R selects one of them [van87].

Definition 3.16: *Let $\leftarrow$<Y,N> be the goal. Let P be the atom selected by a lookahead-cheap computation rule. Let θ be the substitution resulting from the resolution of P by the normal derivation, by the FCIR or by the LAIR. Let I be the set of atoms introduced by the resolution of P. I is the empty set when the LAIR or the FCIR are used and is the body of the selected clause otherwise. Let YI be $Y \cup I \setminus \{P\}$ and NY be the set of atoms Q in N such that $Q\theta \neq Q$. The new goal is*

1. $\leftarrow$ *<$YI \cup NY$, $\{P\} \cup N \setminus NY$>$\theta$ if P is lookahead-available and $P\theta$ contains more than one d-variable;*

2. $\leftarrow$ *<$YI \cup NY$, $N \setminus NY$>θ otherwise;*

where <X,Y>θ is the application of the substitution θ to all the atoms of X and Y [4] [van87].

The implementation of LAIR is quite similar to that of FCIR. However, several recursive calls to the interpreter may be necessary in order to eliminate a single value. Moreover, if the constraint is not solved by LAIR it is delayed again to wait for more information. The pruning of search space in LAIR can be achieved in two ways: first, to restart the computation from scratch to determine which value can be kept in each domain;and second, to compute all the compatible tuples at the first call and in subsequent call to the same predicate, a new tuple is selected as the next answer. Both techniques are implemented in CHIP. Users can choose either of the two by changing a specific parameter in the interpreter.

3.3.3. Partial Looking Ahead Inference Rule

Partial Looking Ahead Inference Rule (PLAIR) provides a formal basis for the implementation of some built-in predicates. It is not used in the same way as FCIR and LAIR in the sense that it does not solve a constraint completely. Hence, it is not implemented as a general method. Nevertheless, the rule is formulated to justify the

[4] See [van87] for the justification of this rule

implementation techniques of some built-in predicates.

Definition 3.17: *PLAIR is defined similarly as LAIR except point 2 to 5 of LAIR have been changed:*

1. *A_m is lookahead-checkable and $x_1, \ldots, x_n$ are the lookahead-variables which range respectively on $d_1, \ldots, d_n$;*

2. *for each $x_j, e_j \supset \{v_j \in d_j$ / there exist $v_1 \in d_1, \ldots, v_{j-1} \in d_{j-1}$, $v_{j+1} \in d_{j+1}, \ldots, v_n \in d_n$ such that $A\theta$ is a logical consequence of P with $\theta = \{x_1/t_1, \ldots, x_n/t_n\}\} \neq \emptyset$;*

3. *z_j is the constant c if $e_j = \{c\}$ or a new variable ranging over e_j;*

4. *$\theta_{i+1} = \{x_1/z_1, \ldots, x_n/z_n\}$;*

5. *G_{i+1} is $\leftarrow (A_1, \ldots, A_k)\,\theta_{i+1}$* [van87].

The set e_j is not defined in this inference rule and is dependent on the implementation of each particular constraint. The soundness of the PLAIR follows from the soundness of LAIR. Plookahead declarations are introduced for the sake of completeness. They are defined as follows:

Definition 3.18: *Given n-ary predicate p, a plookahead declaration, which is unique for this predicate, is an expression of the following form*

plookahead $p(A_1, \ldots, A_n)$

where the A_i's are either g, da, or dp. p is a constraint and all the atoms having p as predicate are said to be submitted to this plookahead declaration [van87].

The semantics of the rule is given by the following definitions:

Definition 3.19: *An atom P submitted to a plookahead declaration is plookahead-available iff*

1. *all the arguments of P corresponding to a g in the plookahead declaration are ground.*

2. *P is lookahead-checkable* [van87].

Definition 3.20: *A computation rule R is plookahead-consistent iff an atom submitted to a plookahead declaration is selected by R only when it is either ground or plookahead-available* [van87].

Definition 3.21: *A proof-procedure is plookahead-consistent iff*

- *it uses a plookahead-consistent computation rule R.*
- *When an plookahead-available atom P is selected by R, the PLAIR is used to solve P. For each lookahead variable x_j, the e_j in the definition of the PLAIR is defined as the domains of x_j when this variable corresponds to a dp in the declaration and as the set obtained for this variable if the LAIR would have been used when this variable corresponds to a da in the declaration. da corresponds to active d-variables while dp corresponds to passive d-variable;*
- *When an atom P that is not plookahead-available is selected by R, normal derivation is used* [van87].

3.4. Built-In Predicates

A significant portion of CTs in CHIP appears as built-in predicates. These predicates perform many functions such as instantiation of variables and solving constraints using specialized methods. In this section, all of CHIP's built-in predicates are discussed. They are grouped together according to their functions.

3.4.1. Variables Instantiation

This category of predicates allows a user to use different strategies to instantiate variables. Several of them are implemented in CHIP: **indomain(X)**, **deleteff(X,L,Lr)**, and **deleteffc(X,L,Lr)**.

The indomain(X) predicate generates values for domain variables. It behaves like the predicate member(X,Lx) where Lx is the list of elements in the domain of X. Each call to indomain(X) gets X bound to a value in its domain. The deleteff(X,L,Lr) predicate selects a variable from the list of variables L. Lr is L with X removed. The criterion for selection is that X has the smallest domain among other variables in L. This kind of implementation follows the first fail principle. Using this predicate in the generator of values, the interpreter can detect failures earlier because the selected variable is more constrained than the others. The deleteffc(X,L,Lr) predicate is an enhanced

version of deleteff(X,L,Lr). The only difference between them is that when the domain sizes of two variables are the same, deleteffc will select the one that involves more constraints.

3.4.2. Specializations Of Inference Rules

Contrary to the three general inference rules, specializations of inference rules allow the user to utilize stronger methods. The idea is to implement a constraint more efficiently by making use of additional knowledge about the constraint. The inequality serves well to illustrate this point. In conventional Prolog systems, the inequality constraint is usually defined as follows:

$X \neq Y \leftarrow$ not $(X = Y)$;

However, this implementation requires that both X and Y are ground for the predicate to be executed correctly. Hence, it can serve as a test only. In CHIP, the constraint can be implemented as a predicate submitted to forward checking:

forward $d \neq d$.

$X \neq Y \leftarrow$ not $(X = Y)$;

If X and Y are d-variables, the declaration imposes the precondition that either X or Y is ground before it can be executed. It is possible to use the general method of FCIR. It is, however, obvious that the constraint can be solved by removing the value of the ground term from the domain of the forward variable. Hence, the following three cases should be used to implement the constraint:

1. If X is a domain variable ranging over D and Y is a natural number, then let Dnew be $D \setminus \{Y\}$. If Dnew = {e}, then X is bound to e; otherwise, X is bound to a new variable Z ranging over Dnew. In both cases, the constraint succeeds;
2. The case where Y is a d-variable and X is an integer can be handled similarly;

3. If both X and Y are natural numbers, the constraint succeeds if X and Y are different integers and fails otherwise.

In fact, there are quite a few built-in predicates that are implemented as specialization of special inference rules. Their discussion is deferred to the next chapter.

3.4.3. Branch And Bound

Branch and bound is a common technique for solving combinatorial optimization problems over natural numbers. A problem using branch and bound consists of two processes [van87]:

- a branching process that splits the problem into several subproblems;
- a bounding process that finds an evaluation bound of a problem.

This approach avoids explicit enumeration of the entire search space. As soon as a solution has been found, all the problems with an evaluation bound not as good as the cost of the solution will not be considered for the branching process. Two built-in predicates are implemented in CHIP to provide a kind of depth-first branch and bound in logic programming [van87]: **minimize(Term, Function)** and **minimize_maximum(Term, List)**.

minimize(Term, Function) instantiates Term such that the value of Function is minimized. The algorithm consists of several steps.

1. A feasible solution is found; otherwise, failure is reported.
2. The new constraint of the form *Function* < *C* is added to the problem. C is the value of Function in the current solution. All future solution must have a cost which is less than C.
3. Step 2 is repeated until the solution space has been searched through.

A similar predicate, **minimize(Term, Function, Lower, Upper)**, is also defined. The third parameter, Lower, is a natural number representing the lower bound of any solu-

tion. It sets a lower bound for the search in that as soon as a solution of this cost is found, the search can terminate since no better solution can be found. The value of Lower is given by the user. There are many methods that can be used to compute such value. For instance, relaxation is a commonly used method. The fourth parameter, Upper, is a natural number representing an upper bound of all solutions. Any solution must have a cost less than this bound and can be used to restrict the search.

minimize_maximum(Term, List) instantiates List in such a way that the value of the largest element of List is minimized. The constraints that will be generated dynamically are of the form $x_i < C$ ($1 \leq i \leq n$) where x_i is the ith element of List which is of size n and C is the value of the largest element in the list. In other words, these new constraints require that the new solution will give a smaller value for C.

3.4.4. The Delay Predicate

The implementation has a simplified delay mechanism to facilitate some problems that involve subgoals which have to be delayed. A predicate that uses the delay mechanism has to be submitted to a delay declaration which is defined as

delay $p(a_1, \ldots, a_n)$

where a_i can be either '+' or '-'. The predicate p can only be selected for execution when each of the arguments that has a '+' in the delay declaration is ground.

3.5. Summary

This chapter has described the implementation of CHIP. We are now in the position to evaluate this implementation. First, CHIP is based on a largely modified Mu-Prolog which has a built-in delay mechanism that allows the implementation of special inference rules. Such a mechanism is not usually found in conventional Prolog

systems. Although there are several limited forms of delay mechanism in some Prolog systems, such as the freeze predicate in PrologII, they are inadequate for the special inference rules. For CTs to be more easily available in logic programming, they should be implemented in conventional Prolog systems. Unfortunately, the kind of data structure that are required to embed CTs into a conventional system is not yet known. Second, CHIP does not use any memory saving technique such as Last Call Optimization. It is possible to incorporate these techniques into an interpreter that use CTs so that the overall efficiency is improved. An implementation of CTs in a conventional Prolog interpreter will provide some insight in this context. In the following chapter, we attempt to answer these questions with a detailed description of our implementation — WUP3.F.

Chapter Four
WUP3.F

This chapter discusses WUP3.F, an implementation of consistency techniques in WUP. It begins with an overview of WUP and proceeds to explain various features of WUP3.F.

4.1. Overview of WUP

WUP is a logic programming environment developed at University of Waterloo [Che84]. It supports the concept of modules and incorporates several memory saving features in its interpreter. These features are highlighted in the following subsections.

4.1.1. Memory Organization

An object in WUP is represented by a PC_WORD which is a record of the following form:

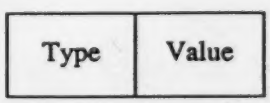


Figure 4.1: A PC_WORD

The first field, Type, indicates the type of the object. The second field, Value, contains either the value of a constant or a pointer which references a block of PC_WORDs that represents a structured object.[1]

[1] A diagram showing all types of objects in WUP is given in Appendix A1.

Similar to most Prolog interpreters, WUP uses three stacks to maintain its execution state: the runtime stack, the copy stack, and the trail stack. It differentiates deterministic computations from non-deterministic ones; therefore, there are two types of stack frames in the runtime stack, as shown in Figure 4.2.

Deterministic

CALL	→Pure Code
FATHER	→Stack Frame
MODULE	→Module

Non-Deterministic

CALL	→Pure Code
FATHER	→Stack Frame
MODULE	→Module
CL_GEN	→Pure Code
BACK	→Stack Frame
RESET	→Trail Stack
COPY	→Copy Stack

Figure 4.2.: Two Types of Stack Frames

The module pointer is used to remember the module that contains the matched clause of the current subgoal. This pointer is used in the management of modules. The CL_GEN pointer is similar to the NEXT CLAUSE pointer in section 1.6. It points to the next available candidate clause for the current subgoal. Moreover, the runtime stack is placed above the copy stack so as to avoid dangling pointers [Hog84].[2]

Since WUP uses the structure copying strategy, the copy stack is mainly used to

[2] Another measure to avoid dangling pointers is that whenever two variables are bound together, a pointer always originates from the higher one to the lower one. These two measures ensure that pointers in the runtime stack always point downwards.

store dynamically constructed terms at runtime.[3] A new object is created on the copy stack under the following circumstances [Che84]:

- during unification: whenever a free variable binds to a structured object that contains variables, a new copy of the object is created on the copy stack and the free variable binds to a pointer that points to the new object;
- whenever there is a structural conversion, e.g. functor to list, the final object is temporarily saved on the copy stack;
- execution of some meta predicates requires that some internal objects be created on the copy stack.

Upon failure of a subgoal, its stack node and the portion of copy stack that are above the COPY pointer of the stack frame are popped.

The trail stack stores pointers that reference the runtime or copy stack. During unification, the addresses of newly bounded variables are recorded on the trail stack. Upon backtracking, these variables are reset to "undefined". Because of Prolog's left-to-right depth-first execution order and its stack-based organization, only those variable bindings in the environment that are below the most recent backtracking node's environment are reset [Che84].

4.1.2. Unification Algorithm

The unification algorithm of WUP is based on [Rob65]. It does not perform the occur check for the sake of efficiency. When infinite structures are unified, the algorithm will loop until all the available storage is exhausted [Che84]. Moreover, the algorithm is table-driven. This feature allows WUP to accommodate new types of objects easily. A new type of objects can be added into WUP by expanding the unification table. The expansion involves only those cases for the unification between an instance

[3] WUP uses structure copying strategy instead of structure sharing strategy as its earlier versions used the latter strategy [Che84].

of the new object and a term.[4]

4.1.3. The Interpreting Algorithm

The interpreting algorithm of WUP is a modified ABC algorithm [van82]. The latter performs a depth-first, left-to-right traverse of a tree to locate a terminal node which has property P. Similarly, the interpreter algorithm of WUP performs a depth-first, left-to-right traverse of the SLD tree to locate a terminal node which is an empty goal. A simplified version of the algorithm [van82] is given below:

[4] The unification table is given in appendix A2.

```
    initialize the stack as empty
    curr-proc := initial goal statement
    { disguised as procedure goal ← ... }
    curr-env := create-env(cur-proc)
    push curr-frame
A:  if select(curr-call)
    then { the current goal statement is non-empty;
           curr-call is the selected goal }
           next-clause := create-cg(curr-call)
           goto B
    else halt with success
    fi
B:  if son(next-clause, curr-call, curr-env, new-frame, curr-proc)
    then
         push curr-frame
         curr-env = ENV(new-frame)
         goto A
    else goto C
    fi
C:  if stack has only one frame
    then halt with failure
    else top-frame := top of stack
         curr-frame := FATHER(top-frame)
         curr-env := ENV(curr-frame)
         curr-call := CALL(top-frame)
         curr-proc := PROC(curr-frame)
         next-clause := NEXT-CLAUSE(top-frame)
         undo bindings of RESET(top-frame)
         pop stack; goto B
    fi
```

The algorithm consists of three sections. Section A deals with the selection of the current subgoal. The computation rule is defined in the function *select*. Section B deals with the matching of the selected goal with the head of a program clause. If there is a successful match, a new stack node is pushed on top of the current one and the computation continues; that is, it will go back to section A. Section C deals with backtracking.

4.1.4. Modules

In WUP, the source code of a logic program can be distributed into different modules. Each module usually contains a collection of definitions of predicates that provides a particular problem-solving technique or supports a specific data structure. For instance, the module *tree* may contain predicates to traverse a tree or to insert a node. This scheme not only allows separate compilation of different modules but also keeps large programming projects manageable as different sections of the source code are stored in different modules.

4.1.5. Memory Saving Techniques

Besides differentiating deterministic computations from non-deterministic ones, WUP uses three memory saving techniques. First, upon backtracking, all the stack frames above the current backtracking point are popped, as all of them are deterministic. Second, WUP performs Last Call Optimization(LCO) [Hog84]. The basic idea of LCO is to detect situations in which recursion can be turned into iteration. LCO will reuse the same stack node of the last call. Three conditions must be satisfied before LCO is applied [Che84]:

- the call involved must be the last call in the clause to which it belongs;
- in the same clause, there must not be any backtracking point between the first call and the call involved;
- the size of the new environment is the same as the old one.

Third, program clauses are indexed on the first arguments of their heads. Given a subgoal, its first argument is used to locate the first and the next candidate clauses. There are three possible outcomes: first, if both clauses are found, the computation is non-deterministic; second, if only one candidate clause is found, the computation is

deterministic; and third, if no candidate clause is found, failure is reported. The scheme avoids a lot of shallow backtrackings and detects deterministic computations earlier.

4.2. Finite Domains

In WUP3.F, only finite domains of integers are defined. We believe that the implementation of domains of integer is sufficient to illustrate the impact of adding consistency techniques into a conventional Prolog system because these techniques are used for discrete combinatorial problems, which are involved mostly with sets of integers.

4.2.1. Internal Representation

A domain is implemented as a set which is represented by a record containing four fields:

- lower - the lower bound of the set;
- upper - the upper bound of the set;
- count - the cardinality of the set;[5]
- setmask- the boolean array that tells which elements are in the set.

The setmask is implemented as a sequence of bytes. Each bit in the sequence acts as a marker that indicates whether the corresponding element is in the set or not. The mapping between the elements of the domain and the setmask is similar to the one mentioned in section 3.2.

To facilitate efficient computation of the intersection of two domains, each setmask is aligned. By alignment, it means that a setmask always starts at the greatest

---

[5] The cardinality of a set is always greater than one. A d-variables whose domain is of cardinality 1 will be instantiated as the value of the remaining element.

multiple of 8(i.e. the size of a byte) that is less than the lower bound of the set, and ends at one less than the least multiple of 8 that is greater than the upper bound of the set. For instance, if a domain ranges over from 5 to 102, its setmask should start at 0 and end at 103(i.e. 1 less than 104). Thus, the intersection of two sets can be computed using the binary AND for each corresponding pair of bits in the setmasks. For instance, given two sets {1, 3, 5, 7} and {2, 3, 4, 5}, their setmasks will be

01010101 and
00111100

and the intersection will have the setmask

00010100

which represents the set {3, 5}.

4.2.2. Domain Declaration

The domain declaration is implemented as a built-in predicate of the following form:

?domain @Dp($a_1, \ldots, a_n$)

@D is a tag which indicates that predicate p has a domain declaration. Each a_i is a domain specification (see section 3.2).

A domain declaration is usually included in the source file of a logic program. When consulting the file, the interpreter will execute this domain declaration as a built-in predicate. It first checks whether each domain specification is valid or not; an invalid specification will cause the entire declaration to be ignored. Then, the interpreter asserts the predicate @Dp($a_1, \ldots, a_n$) into the module *domain*.

At runtime, whenever a subgoal with the tag @D in its name is selected, the interpreter searches the domain module to locate the subgoal's domain declaration.

Then, the arguments of the subgoal and those of the declaration will be compared in a pairwise fashion. If the domain specification of the current argument is 'h', nothing is done because 'h' is compatible with any term. Otherwise, the specification defines a domain. The interpreter will construct the domain using the given specification, and every free variable in the subgoal's argument is instantiated to be a d-var with the constructed domain.[6] For every non-variable term, the interpreter checks whether it is a member of the domain; if any one of them is not a member of the domain, failure is reported and backtracking occurs.

4.2.3. Unification of Finite Domains

The unification table of WUP is expanded in order to handle the unification between a d-variable and a term. Assume that X is a d-var; the unification between X and a term Y can be summarized in three cases:

- If Y is a d-var, the intersection of the domains of both variables is computed. If the intersection is not empty, both X and Y will be bound to a new d-var whose domain is the intersection. Otherwise, unification fails.
- If Y is an integer, X is bound with Y if Y is a member of the domain of X. Otherwise, unification fails.
- If Y is a free variable, Y is instantiated to be a pointer to X.

4.3. A Model for the Delay Mechanism

The implementation of the special inference rules in a conventional Prolog system requires a computation rule that is similar to that of Mu-Prolog. This computation rule must ensure the followings:

- if the precondition for execution of a selected subgoal is not satisfied, the subgoal is delayed;

---

[6] Every variable gets an independent copy of the domain.

- once its precondition for execution is satisfied, a delayed subgoal should be activated;
- each delayed subgoal should only be executed once because multiple executions cause incorrect computations.

In section 4.2, the computation rule of WUP3.F is discussed. It is based on the work of Boizumault [Boi86] though there are quite a few modifications.

4.3.1. Boizumault's Model

The model of Boizumault explains the structure of a conventional Prolog interpreter that can execute the freeze predicate [Boi86]. This predicate gives the interpreter the ability to postpone the execution of a subgoal when one of its arguments is not instantiated. In short, the model allows a restricted form of delay mechanism to be defined.

To implement the freeze predicate, the interpreter is modified in four aspects. First, a new type of variables, called the frozen variable, has to be defined. These variables are free variables and each of them associates with a list of frozen subgoals, i.e. those delayed subgoals. When a subgoal is delayed, it is linked to the frozen variable that causes the delay. Second, a new stack, called the frozen goals stack, is added. It is used to store delayed subgoals which are maintained in the form of push-down lists [Boi86]. Third, a value trail[7] must be used to manage the push-down lists of frozen subgoals. If the trail stack of the current system is not a value trail, it has to be modified. Finally, the unification must be expanded to allow unification between a term and a frozen variable. Assuming that X is a frozen variable and Y is a term, unify(X,Y) covers the following cases:

---

[7] Each node in the trail contains two values: the address of a term that gets the new binding and the value of the old binding. Upon backtracking, the term is assigned the value of the old binding.

- if Y is a non-variable term, then X is bound to Y and the list of waiting goals of X is activated;
- if Y is a free variable, Y is bound to a pointer that references X;
- if Y is also a frozen variable, both lists of waiting goals (of X and Y) are merged to form a new list. The two variables are bound to a new frozen variable which is associated with the new list.

Moreover, the algorithm of the freeze predicate, i.e. freeze(X,Y), is defined as follows [Boi86]:

```
t = the dereferenced value of X
if (t is a non-variable term)
     Y is proved in the normal way
else if (t is a free variable) {
     X binds to a frozen variable whose list of frozen goals
     contains Y as the only element
}
else /* t is also a frozen variable */
     Y is appended to X's list of frozen goals list
```

This model is simple because each frozen subgoal associates with one frozen variable. Once the variable gets instantiated, the associated subgoals can be executed. Because of this one-to-one relationship, the data structure that manipulates frozen subgoals can just be a simple linked list. In addition, no extra code is needed as the above algorithm can be implemented as a built-in predicate. In other words, the modification that is required of an interpreter to use freeze is rather small. However, this model is not sufficient to be used as a general delay mechanism because of two reasons. Firstly, the freeze predicate defines a restricted form of preconditions which allow only the delay of a subgoal when one of its argument is not instantiated. It is not general enough to define more sophisticated preconditions such as the condition of forward available for forward constraints. Secondly, the model does not use any declaration. A predicate, for example p(X), that uses the freeze predicate must make an explicit call, i.e. freeze(X,p(X)). Yet the use of declarations allows different preconditions to be

defined for predicates that use the same inference rule. In view of the deficiencies of this model, a general model is proposed and developed.

4.3.2. A New Model

The model proposed here is an extension of Boizumault's model. It allows more sophisticated preconditions to be defined. Moreover, it can handle different types of declarations. The model is implemented in WUP3.F to provide a computation rule that is very similar to the computation rule in CHIP. It consists of three major components: a checking routine, an expanded unification algorithm, and an activation mechanism.

4.3.2.1. The Checking Routine

The checking routine determines if a subgoal should be delayed or not. It locates the declaration of the subgoal and checks whether the subgoal is compatible[8] with its declaration. The subgoal will not be delayed if it does not have a declaration or if it is compatible with its declaration. The algorithm of the checking routine, called Check-Delay, is defined as follows:

```
/* cur_goal is the current subgoal */

d = GetDeclaration(cur_goal)
if (d is nil) then return(FAILURE) /* if no declaration, don't delay*/
if ( ! Compatible(cur_goal,d)) {
      Delay(cur_goal)
      return(SUCCESS)
}
else
      return(FAILURE)
```

[8] A subgoal is compatible with its declaration if it satisfies the precondition of execution defined by its declaration.

The data structure used in the algorithm is a linked list of *delayed calls* shown in figure 4.3:

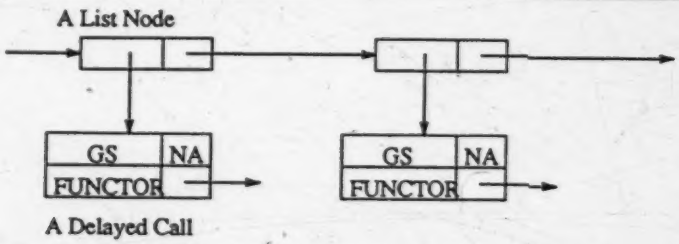


Figure 4.3: A Linked List of Delayed Calls

Each element in the linked list is called a list node. It has two pointers: one points to the next node in the list and the other points to a delayed call. A delayed call contains a FUNCTOR field and a STATUS field. The FUNCTOR field contains a copy of the subgoal that is being delayed. The STATUS field is a flag that indicates whether the delayed call has been activated or not. It is used in the activation mechanism (see section 4.3.2.3.).

When a subgoal is delayed, a delayed call is created in the copy stack. The STATUS field will be marked NOT_ACTIVATED to indicate that this delayed call is not activated. For each variable in the subgoal that falsifies the precondition defined in the declaration, a list node is allocated in the frozen goal stack and a link is created between the node and the delayed call. For instance, if two variables falsify the precondition, two links will be set up. An example will best illustrate how this component works.

Consider the goal ?p(X,Y), q(X,Y,Z) ... where X, Y, and Z are free variables. Furthermore, p(X,Y) and q(X,Y,Z) have the delay declarations p(+,+) and q(-,+,+), respectively. This means that p can be executed if X and Y are ground and that q can be

executed if Y and Z are ground. In the beginning of the execution, p(X,Y) is selected as the current goal. The checking routine then detects that p(X,Y) should be delayed because both X and Y are not instantiated. A delayed call is allocated in the copystack for p(X,Y); and two list nodes are allocated in the frozen goal stack, one for each of the variables. In addition, two links are set up, each between the delayed call and a list node. X and Y become frozen variables and each of them has the delayed call, p(X,Y). In the runtime storage, the following scenario is observed:

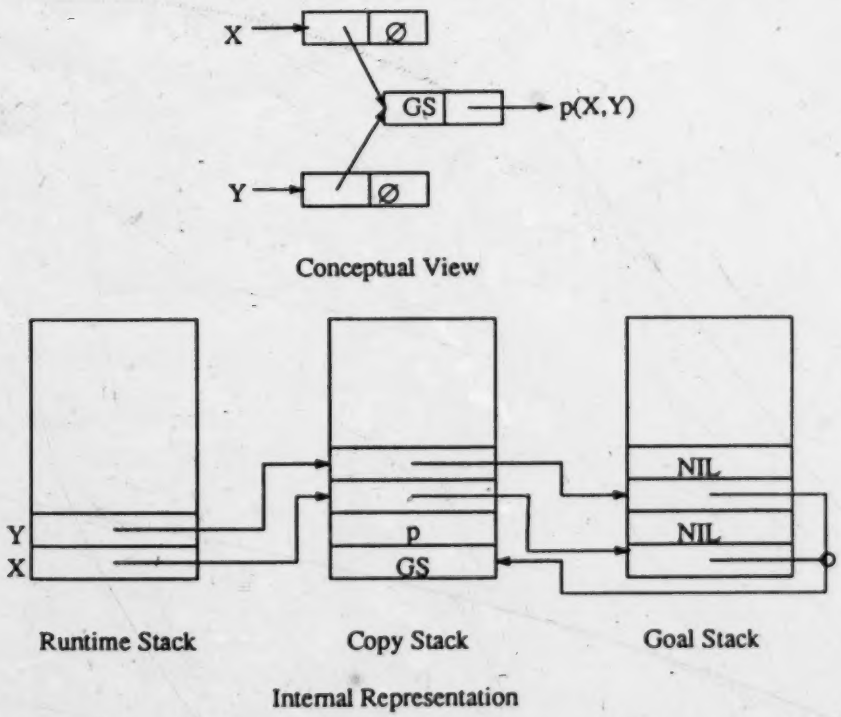


Figure 4.4: Two Views of the Delay of p(X,Y)

After p(X,Y) is delayed, q(X,Y,Z) will be selected as the current subgoal. The checking routine detects that q should be delayed as both Y and Z are not instantiated. A delayed call for q(X,Y,Z) and two list nodes are created, one for Y and the other for Z. More-

over, appropriate links are established between the two nodes and the delayed call. Since Y already has a list of delayed calls, i.e. p(X,Y), the new list node must be inserted to the existing list to form a new list. The new list contain two delayed calls: q(X,Y,Z) and p(X,Y). The following scenario is observed:

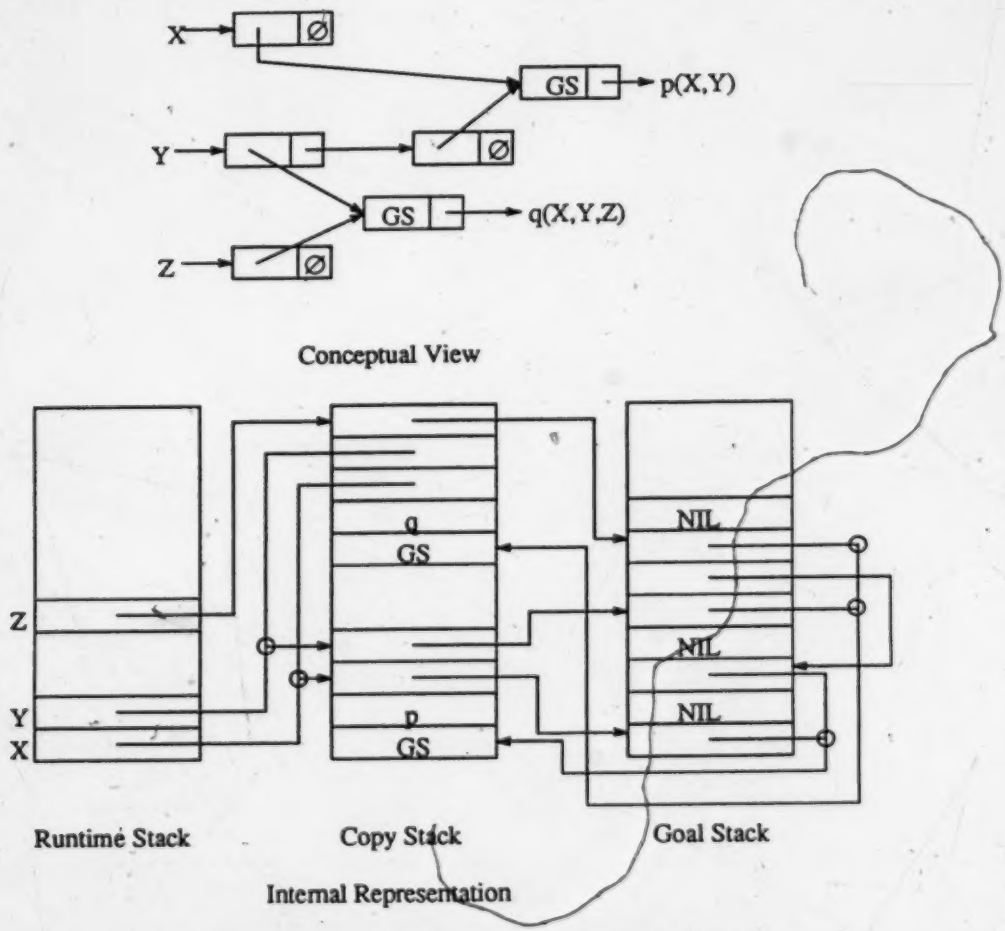


Figure 4.5: Two Views of the Delay of p(X,Y) and q(X,Y,Z)

4.3.2.2. The Expanded Unification Algorithm

The second component of the delay mechanism is the modified unification algo-

rithm. The modification required is very similar to that of Boizumault's model. One major difference lies in the use of the GoalList stack. Whenever a frozen variable get instantiated, the address of its list of delayed subgoals is pushed onto this stack. These subgoals are not activated until an inference[9] is successfully completed because it is possible that failure may occur before the inference is completed. Moreover, there may be more than one frozen variables being instantiated during the inference. The GoalList stack keeps track of the addresses of all these variables.

4.3.2.3. The Activation Mechanism

The third component of the modified model is the activation mechanism. A delayed call may link to more than one of its variables. The use of the STATUS flag can eliminate the chance of activating the same delayed call more than once. Whenever a delayed call is activated, the status flag is assigned the value ACTIVATED and this change is trailed. Later on, if the interpreter has to activate the same delayed call again, the flag will tell whether the activation is necessary or not. Therefore, even if two of the variables of a delayed call get instantiated, the delayed call will be activated only once.

When a frozen variable gets instantiated, some of the subgoals in its list of delayed calls may satisfy their preconditions for execution. These subgoals should be checked again. However, this step is very expensive. A feasible way is to reactivate every waiting goal (i.e., to insert them into the current resolvent) and let the execution continue. Thus, the interpreter will eventually select one of these goals and will check its precondition. This method involves a lot of repeated delays and activations but it is difficult to have greater efficiency with the generality of wait declaration [van87].[10]

---

[9] An inference refers to the unification between a selected goal and the head of a program clause. Usually, the unification is recursively performed for each pair of corresponding arguments of the two atoms.

WUP3.F uses this method.

After an inference is successfully completed, the entries in the GoalList stack are used to locate all the subgoals that are pending for activation. These subgoals are linked together to form a conjunction. The conjunction is then inserted into the current resolvent. After this, the execution is continued.

The activation mechanism must be incorporated into the interpreting algorithm. In the case of WUP3.F, the section A of the ABC algorithm is modified to:

```
select_again:
    if ( GoalListStack <> nil ) /* there are goals to be activated */
        curr_call = ActivateGoals(GoalListStack)
    while (curr_call = nil) {
        curr_frame = FATHER(curr_frame)
        if ( curr_frame = nil )
            return(SUCCESS)
        else
            curr_call = next_call(CALL(curr_frame))
    }
    if ( CheckDelay(curr_call) ) { /* if curr_call is delayed */
        curr_call = next_call(curr_call) /* go to next subgoal */
        goto select_again
    }
    curr_proc = PROC(curr_frame)
    curr_env = ENV(curr_frame)
```

The function ActivateGoals inserts all the delayed subgoals that have to be activated into the current resolvent. To understand how the activation mechanism works, consider the following program:

---

[10] The general model actually implements a variation of the wait declaration.

```
A ← B;
C ← D;
E ← F & G & H;
? A & C & E;
```

Assume that B and D are delayed after A and C are executed and that if E is executed, A and C can be activated again. So before the execution of E, the resolvent is ?E. However, the resolvent becomes ?B & D & F & G & H just after an inference on E is made, i.e. B and D are inserted before F.

4.4. Special Inference Rules

This section discusses the implementation of both Forward Checking Inference Rule (FCIR) and Looking Ahead Inference Rule (LAIR) in WUP3.F. It first shows the declaration and the proof procedure of each rule. Then, it explains how the rules, together with the delay predicate, are integrated into WUP. The Partial Looking Ahead Inference Rule (PLAIR) is not discussed here because the rule does not have a general proof procedure, and each predicate submitted to PLAIR has to perform its own pruning.

4.4.1. Forward Checking Inference Rule

In WUP3.F, a constraint that uses FCIR must be submitted to a forward declaration of the following form:

?forward @Fp(a_1 , . . . , a_n)

where a_i are either 'g' or 'd'. When the interpreter executes a domain declaration, the declaration @Fp(a_1 , . . . , a_n) is asserted into the module *forward*. The tag @F is used to indicate that predicate p has a forward declaration. This reduces the time required to

search for the declaration.

The proof procedure of FCIR is summarized as in the following algorithm:

```
Vf = forward variable
S = DomainOf(Vf)
Snew = ∅
SaveInterpreterState()

/* test each value of the domain to see if it is consistent */

for each k ∈ S {
      bind(Vf, k)
      if prove(p) Snew = Snew ∪ {k}
      reset(Vf)
}

RestoreInterpreterState()
if (Snew <> ∅) {
      S = Snew
      return(SUCCESS)
}
else
      return(FAILURE)
```

This algorithm follows closely the description in section 3.3.1. prove(p) is a recursive call to the interpreter and it has to use a clean interpreter. Therefore, the function SaveInterpreterState() is called before prove(p) is executed. It saves the values of all the important variables in the interpreter temporarily. After the proof is done, the original state of the interpreter can be restored by calling RestoreInterpreterState() which restores the variables whose values are saved previously.

The proof procedure of FCIR is not used to solve all forward constraints. In particular, two types of forward constraint use other methods. First, if the constraints are built-in predicates, they will use more efficient methods to perform the pruning (see section 4.5.). Second, a ground constraint does not have any d-variable; consequently, no pruning is required and normal derivation is used.

4.4.2. Looking Ahead Inference Rule

LAIR is a generalization of FCIR [van87]. When it is used, all the inconsistent values in all d-variables of a lookahead constraint can be eliminated. However, the proof procedure is more complicated than that of FCIR. Usually, several calls are required to eliminate a single value from the domain of a d-variable. As the sizes of domains and number of variables increase, LAIR becomes very expensive.

To use LAIR, a constraint must be submitted to a lookahead declaration of the following form:

?lookahead @Lp($a_1, \ldots, a_n$)

where each of a_i can be either a 'g' or a 'd'. The tag @L indicates the predicate p has a lookahead declaration. When the interpreter executes a lookahead declaration, it asserts the declaration @Lp($a_1, \ldots, a_n$) into the module *lookahead*. If a subgoal with the tag @L is selected, the interpreter will search for its declaration in the lookahead module. The checking routine will then determine whether it is lookahead available or not.

There are two methods to perform the pruning in LAIR [van87]. The first one is to generate all the tuples that are valid for the constraint and store them in the form of a list in the copy stack. When the interpreter backtracks to the same predicate, the next element to the right of the current tuple can be selected as the new solution. The second method is to do the computation from scratch each time the constraint is executed. Inconsistent values in the d-variables are deleted during the computation. This method does not keep track of all the valid tuples for the predicate. As there are no definite advantage and disadvantage between these two method, WUP3.F uses the second method because it is easier to be implemented. The algorithm for the second method is described as follows:

```
/*
   p(t1, . . . , tk,v1, . . . , vm) is the constraint to be solved;
   t1, . . . , tk are ground terms in p; v1, . . . , vm are d-variables in p;
   D1, . . . , Dm are domains of vi's;
   x1, . . . , xm are temporary variables, one for each vi;
   E1, . . . , Em are the temporary domains for vi's;
*/
SaveInterpreterState()
for i from 1 to m  Ei = ∅

for i from 1 to m {          /* i.e. for each d-variable vi*/
   while (Instantiate([Di], [xi])) { /* instantiate xi from its domain */

      if (xi ∈ Ei) continue; /* skip this value of xi */

      solved = false

      /* instantiate other d-variables  */

      while ( not solved and
         Instantiate( [E1, . . . , Ei-1,Di+1, . . . , Dm],[x1, . . . , xi-1,xi+1, . . . , xm])) {
         solved = prove(p(t1, . . . , tk,x1, . . . , xm))

         if (solved)
            for j from i to m   /* remember all xi's */
               Ej = xj ⋃ Ej   /* and keep them in Ei's */
      }
   }
}

RestoreInterpreterState()
for i from 1 to m {
   if (Ei <> ∅ )  Di = Ei
   else
      return(FAILURE)
}
return(SUCCESS)
```

The Instantiate(DL,VL) function used in the above algorithm is a generator of values for variables in VL. When Instantiate(DL,VL) is executed, it assigns a new combination of values for the variables in VL. Each v_i is assigned an element in D_i. When all the combinations are tried, the function will fail. For example, Instantiate([{1,2},{1,2}], [v_1,v_2]) will generate [1,1], [1,2], [2,1] and {2,2] for [v_1,v_2].

Similar to FCIR, LAIR is not used for two types of predicates: ground predicates and built-in predicates. However, after the proof procedure is applied once, those constraints that are not solved completely have to be delayed again until more information is available. Those partially solved built-in predicates are also delayed.

For lookahead constraints, a modification in the domain of a d-variable may provide some useful information for further pruning [van87]. These constraints should be activated immediately. However, a d-variable may associate with some constraints other than lookahead constraints. The interpreter should activate only the lookahead constraints. Therefore, two separate lists of delayed subgoal in a frozen variable are kept: one list for lookahead constraints and the other for non-lookahead constraints. When only lookahead constraints have to be activated, the first list of delayed subgoals can be activated.

4.4.3. Integration with the Interpreter

To put finite domains, special inference rules, and the delay declarations together, the interpreter of WUP has to be modified in two ways, in addition to the modifications mentioned in the new model. Firstly, a new type of variables called frozen domain variables must be added into WUP. A frozen domain variable references three objects: its domain and two lists of delayed goals. Each variable of this type is a pointer that references a block of three PC_WORDs; each of which is a pointer to one of the three objects. This design of the frozen domain variables also affects the structures of the domain variables and the frozen variables. Figure 4.6 shows the structures of these three types of variables.

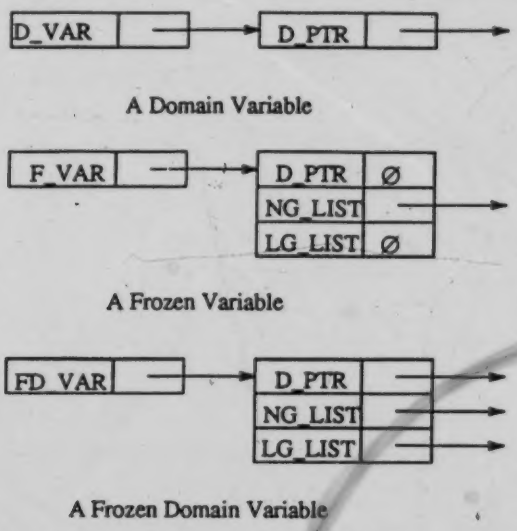


Figure 4.6: Three Types of Variables in WUP3.F

Since a frozen variable has just one list of delayed subgoals, the normal list pointer is used to reference the list and the lookahead list pointer is always assigned nil.

Secondly, the section A of the ABC algorithm must be expanded to allow the use of FCIR, LAIR, and the delay predicate. The following algorithm represents the final version of section A of the algorithm:

```
select_again:
    if ( (GoalListStack <> nil) or (LookaheadGoalListStack <> nil) )
        curr_call = ActivateGoals(LookaheadGoalListStack, GoalListStack)
    while (curr_call = nil) {
        curr_frame = FATHER(curr_frame)
        if ( curr_frame = nil )
            return(SUCCESS)
        else
            curr_call = next_call(CALL(curr_frame))
    }
    n = name(cur_call)
    if (n[1] = '@') {  /* check if there is a tag in the name */
        switch (n[2]) {
            case 'F':  /* for forward constraints */
                if ( decl = GetDeclaration(ForwardModule, cur-call)) {
                    status = ProcessForwardCall(cur-call, decl)
                    switch (status) {
                        case SUCCESS:
                            cur-call = NextCall(cur-call)
                            goto select_again
                        case FAILURE:
                            goto_next_proc
                        case CONTINUE:
                            break
                    }
                }
                break
            case 'L': /* for lookahead constraints */
                if ( decl = GetDeclaration(LookaheadModule, cur-call)) {
                    status = ProcessLookaheadCall(cur-call, decl)
                    switch (status) {
                        case SUCCESS:
                            cur-call = NextCall(cur-call)
                            goto select_again
                        case FAILURE:
                            goto_next_proc
                        case CONTINUE:
                            break
                    }
                }
                break;
            case 'E': /* for the delay predicate whose tag is @E*/
                if ( decl = GetDeclaration(LookaheadModule, cur-call)) {
                    status = ProcessDelayCall(cur-call, decl)
                    if ( status = SUCCESS ) {
                        cur-call = NextCall(cur-call)
                        goto select_again
                    }
                }
```

```
                    break
                default: /* ignore other tags such as @D */
                    break
                }
        }
    curr_proc = PROC(curr_frame)
    curr_env = ENV(curr_frame)
```

This algorithm can be described as follows. Whenever a subgoal is selected as the current subgoal, the interpreter detects whether the subgoal has a tag or not. If a predicate has no tag, nothing is done (i.e., the predicate is a normal one). Otherwise, depending on the type of tag, the interpreter will pass control to a special routine which processes the constraint. There are three possible outcomes after the processing:

SUCCESS
: the constraint is either solved completely or delayed. Execution will continue.

CONTINUE
: this case is for those ground predicates or for those built-in predicates that perform their own pruning.

FAILURE
: the proof procedure fails to prove the predicate and backtracking is needed.

4.5. Built-in Predicates

A large collection of built-in predicates of CHIP are implemented in WUP3.F. These predicates provide efficient means to solve a variety of constraints. As mentioned in section 3.4, they are grouped into three categories: variables instantiation, specialization of inference rules, and the branch and bound method.

4.5.1. Variables Instantiation

The indomain predicate is a generator of values for d-variables. It is implemented like the predicate member(Element, List). In WUP3.F, indomain is defined as follows:

```
indomain(X) ←   /* if X is instantiated, return true */
        is_int(X) &
        cut;
indomain(X) ←   /* otherwise, locate an element from its domain */
        pick_element(X,_,N) &
        pick_next(X,N,L) &
        X = L ;

pick_next(D,L,L);
pick_next(D,L,E) ←
        pick_element(D,L,L1) &
        pick_next(D,L1,E) ;
```

where pick_element(D,L,E) is defined as:

- if L is a free variable, pick_element will assign the smallest (i.e. the first) element of d-variable D to E;
- otherwise, E will be assigned the element which is the least element in D that is larger than L.

The major function of this predicate is to instantiate d-variables. The order of instantiation is defined arbitrarily by the user; however, if extra information about a particular problem is available, the order of instantiation can be defined to allow faster pruning of the search space.

The *deleteff* predicate and *deleteffc* predicate of CHIP are implemented in WUP3.F. In the implementation of deleteffc, the interpreter has to count explicitly the number of delayed calls that a d-variable has.

4.5.2. Specialization of Inference Rules

This group of predicates utilizes additional knowledge about the constraints it represents and uses more direct methods to solve them. They do not use the proof procedures defined by the inference rules.

4.5.2.1. Specializations of FCIR

Inequality constraints appear in lots of CSPs and efficient implementation of them is desirable. In WUP3.F, an inequality constraint is denoted as X ≠ Y. It has the forward declaration @FB≠(d,d). When an instance of the constraint is forward available, it can be proved in two ways:

- if the constraint is ground, the common definition of inequality is used;
- otherwise, one term must be an integer and the other a d-var. The interpreter deletes the value of the integer from the domain of the d-var.

Constraints that represent binary relational operators, such as >=, <= , > and <, can be defined similarly. For instance, when the constraint X > 5 is executed, all the elements in the domain of X that are less than or equal to 5 are removed.

4.5.2.2. Specialization of LAIR

The element(N,L,V) predicate is a very useful symbolic constraint [DSv88]. It holds when the Nth element of the list of constants L is V; both N and V are d-variables. The interesting feature of this constraint is that it can be viewed as an adirectional constraint that makes a correspondence between V and N. This means that a constraint on one of them will have effect on the other (see the example in [DSv88]).

The predicate is implemented as a specialization of LAIR. When either N or V is an integer, the constraint becomes trial. The difficult case occurs when both N and V are d-variables as correspondence between the elements in the domain of V and that of N must be established. The following algorithm is used in WUP3.F:

```
/*
     N_o and V_o are the original domains of N and V;
     N_f and V_f are the domains of N and V after execution;
     L [i] is the ith element of the list L.
*/

while (Instantiate( [N_o], [index])) {
     N_o = N_o - {index}
     element = L [index]
     if ( element ∈ V_o ) {
          N_f = N_f ∪ {index}
          V_f = V_f ∪ {element}
     }
}
```

Beside element(N, L, V), two specializations of LAIR, i.e. min(X, Y, Z) and max(X, Y, Z), are also implemented in WUP3.F.

4.5.2.3. Specialization of PLAIR

Linear equations and linear inequations are solved as a specialization of PLAIR. In WUP3.F, a built-in predicate, reduce(X op Y), is implemented. The arguments X and Y are arithmetic terms and op is one of the operators: =, >, < , ≥ and ≤. The first step to solve a constraint expressed in this fashion is to standardize the expression X op Y. Let $x_1, \ldots, x_n$ be d-variables and let $a_0, \ldots, a_n$ be non-zero integers. It is possible transform an inequation or an equation into the standard form:

$$a_0 + a_1x_1 + \cdots + a_nx_n \; © \; 0$$

where © is either >= or =. For example, X > Y can be transformed as X - Y + 1 ≥ 0.

The second step is to perform the pruning in the domains of d-variables. An algorithm in [Lau78] provides the method of pruning and no new constraints are created dynamically [van87]. When a reduce predicate is selected as the current subgoal, the pruning of d-variables is computed from scratch. A description of the

algorithm is given in appendix A3.

Like those lookahead constraints that are implemented as built-in predicates, a reduce predicate may not be solved completely even after pruning is performed. In this case, the predicate has to be delayed to wait for more information.

4.5.3. Branch and Bound

As mentioned in 3.4, the branch and bound method is a well-known technique for solving concrete combinatorial problems. In WUP3.F, two predicates that employ this particular technique are implemented, i.e. minimize(T, E) and maximum_minimize(T, L). Since both predicates require that new constraints be added during runtime, it is more appropriate to implement them as prolog programs. The simplified listing of minimize(T,E) is shown below:

```
minimize(T,E) ←
      generate_solution(T,E) ; /* find subsequent solutions */

generate_solution(T,E) ←
      last_solution(LE) &      /* get previous solution */
      reduce(LE > E) &         /* enforce minimum condition */
      m_prove(T,E) &           /* find the next solution */
      fail ;                   /* try again */
generate_solution(T,E) ←
      last_solution(E);

m_prove(T,E) ←
      prove(T) &
      assert_solution(E) &     /* remember the new solution */
      cut ;
```

The implementation of the minimum(T,E) is similar to that of the setof predicate in that both predicates remember the solutions obtained during runtime. However, in minimum(T, E), new constraints are added to strengthen the condition LE > E.

4.6. Conclusion

In this chapter, WUP3.F has been discussed in detail. The major components that allow CTs to be implemented in WUP are finite domains and the modified model for the delay of subgoals. These two allow the implementation of special inference rules such as FCIR, LAIR, and some powerful built-in predicates. It is also shown that many of the built-in predicates can be implemented in WUP without much difficulty. In the next chapter, a performance analysis is carried out to show that WUP3.F is of comparable efficiency.

Chapter Five
Performance Analysis

This chapter shows the results of a performance analysis of WUP3.F. These results are obtained from five examples solved in WUP3.F. Several criteria for the interpretation of the result are given. Then, each example is briefly explained and the results are interpreted.

5.1. Interpretation Criteria

To interpret the performance statistics of WUP3.F correctly, several criteria must be used. They are execution time, memory usage, strategies employed and the environment. Since the author does not have access to CHIP, it is not possible to run CHIP[1] on the same machine (i.e. SUN 3/50) that WUP3.F is running on. Nevertheless, according to the technical personnel in Digital Equipment and the Department of Computing Science in this university, the speed of a VAX-11/785 and that of a SUN 3/50 are relatively the same — both of them are 1.5 times as fast as a VAX-11/780. Therefore, one can compare the results from WUP3.F directly with those from CHIP which are reported in [van87].

5.1.1. Execution Time

Execution time is used as the major performance indicator. The standard measure LIPS (Logical Inference per Second) is not used because it is an inadequate measure for WUP3.F. There are three reasons for that. First, in each inference, the selected

[1] CHIP runs on a VAX-11/785.

subgoal has to be checked to see if it has a domain declaration or a declaration of a special inference rule. This check requires a definite amount of CPU time. Therefore, an inference in WUP3.F usually requires more time than that of a conventional Prolog system. Second, when a selected goal does not satisfy its precondition for execution, it has to be delayed. The subgoal right next to the selected subgoal is selected then. Such a delay is not counted as an inference in WUP3.F. Third, much of the solving power for combinatorial problems lies on the implementation of finite domains and various built-in predicates. These predicates not only provide a direct means to solve constraints, but also perform pruning for d-variables effectively. Some of these predicates can be viewed as logic programs. If a call to a built-in predicate is treated as a single inference, the performance statistics will always give a small number of inferences. In short, if LIPS (Logical Inferences per Second) is used, all the performance statistics shown here will be too low to justify CTs in logic programming as a worthwhile approach.

5.1.2. Memory Usage

Normally, a performance analysis of a compiler or an interpreter will consider its memory usage. In WUP3.F, additional memory has to be allocated for the Frozen Goal Stack, the value trail, and the expanded copy stack. Other than these three areas, there is no major memory increase or overhead. Moreover, declarations for domains and special inference rules are stored in four separate internal modules. For a program of average size, the number of clauses in each of these modules is small and the amount of extra memory used is negligible. Therefore, memory usage is not used as a performance statistic.

5.1.3. Strategies Employed

The execution time of a Prolog program depends largely on the strategy used to tackle the problem. Obviously, the more efficient the strategy, the less the execution time. It is improper to compare directly the execution times of two Prolog programs that are written in the same system but use different strategies. In the examples used in this chapter, when different strategies are used, they are explicitly stated.

5.1.4. The Environment

The word "Environment" refers to the Prolog system that is used to solve an example. Several programs, each written in a different Prolog system, are always used to solve a particular problem. This allows a better comparison among WUP3.F and other systems. Besides WUP3.F, three Prolog systems are used in the examples. They are WUP3.1, CLP(R), and SICSTUS [CaW88]. WUP3.1, as mentioned in the previous chapter, is a conventional Prolog interpreter. Programs written in WUP3.1 are always the slowest. This indicates why the CTs approach should be used in logic programming. CLP(R) [HJM87] has been described in chapter two; it is used here to write programs that involve equations as well as inequations. SICSTUS Prolog is a Prolog compiler; it is used to show how well WUP3.F compares to a Prolog compiler in terms of execution time. Even though a Prolog compiler is almost one order of magnitude faster than a Prolog interpreter, in some examples where forward checking greatly reduces the search space, WUP3.F outperforms SICSTUS.

In order to provide a fair comparison among the four Prolog systems, it is necessary to find out the relative speed of each system. Therefore, a calibration is carried to determine the relative speeds of the four different Prolog systems. The N-queens problem is used in the calibration. Only standard Prolog features are used so that any special features that would help to speed up the execution are eliminated. Three programs are

written — one for both WUP3.1 and WUP3.F, one for CLP(R) and one for SICSTUS. These programs are run on the four Prolog systems in a VAX-11/780 and a SUN 3/50. The speed of WUP3.F on a SUN 3/50 is assumed to be unity. Table 5.1 shows the relative speed of each system.

	VAX-11/780	SUN 3/50
WUP3.F	0.49	1.00
WUP3.1	0.65	1.22
CLP(R)	N/A	1.59
SICSTUS	20.0	33.3

Table 5.1: Relative Speeds for Different Prolog Systems

It is clear from Table 5.1 that WUP3.F is the slowest while SICSTUS Prolog is the fastest. The entries in this table is used to adjust all the results. Since the results are produced on a SUN 3/50, only the entries in the rightmost column of the table are used to scale the results obtained in the performance test.

5.2. Examples

In each of the following examples, the problem is first briefly introduced. Then, the execution times of various versions of logic programs are compared. Execution times are in term of seconds unless otherwise specified and only scaled values are reported. The listing of all the programs written for the examples are included in appendix A4.

5.2.1. The N-queens Problem

The N-queens problem is to place n queens on a n x n chess board. It can be

reduced to the assignment of a row in each column of the board. No two queens should be on a diagonal or a horizontal line. Let x_i be the row of the ith column The problem is to find a list of $[x_1, \ldots, x_n]$ such that:

- $1 \le x_i \le N$ for $(1 \le i \le N)$;
- $x_i \ne x_j$ for $(1 \le i \le j \le N)$;
- $x_i \ne x_j + (j - 1)$ for $(1 \le i \le j \le N)$;
- $x_i \ne x_j - (j - 1)$ for $(i \le i \le j \le N)$;

The purpose of this problem is to illustrate how forward checking and the first fail principles can be utilized in WUP3.F. Four different programs are written to solve the problem in three Prolog systems as shown in Table 5.2.

Program Name	System	Strategy
ST1	WUP3.1	standard backtracking
ST2	SICSTUS	standard backtracking
FC	WUP3.F	forward checking
FCF	WUP3.F	forward checking + first fail principle

Table 5.2: Programs Written for the N-queen Problem

All the programs are tested for $5 \le N \le 20$. Since WUP3.1's timing mechanism overflows when $N \ge 16$, execution time for ST1 is not available for $N \ge 16$. Figure 5.1 shows the results of the test.

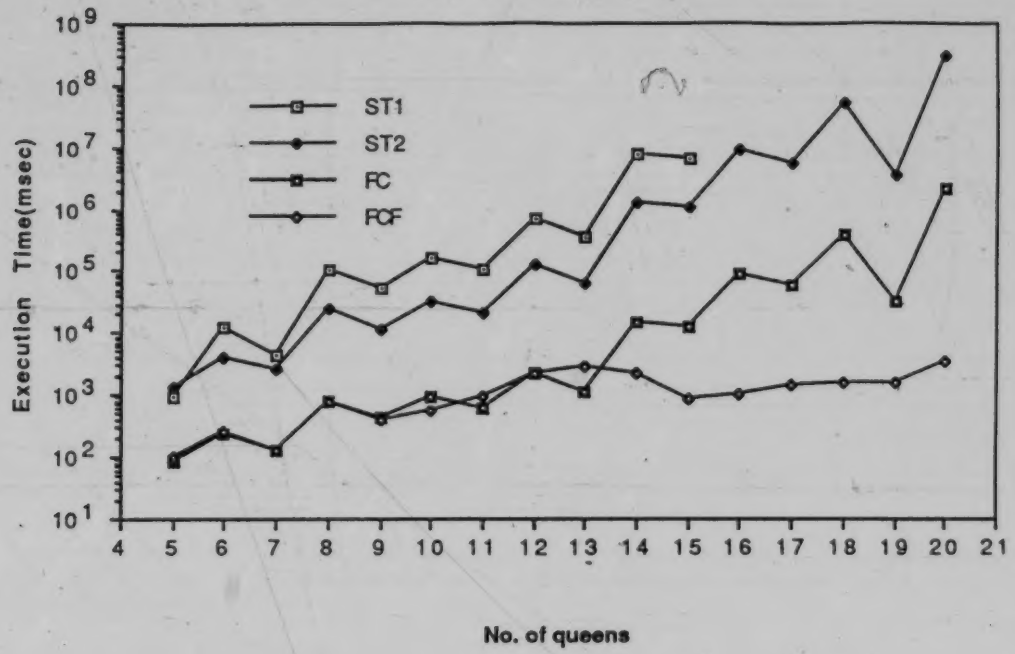


Figure 5.1: Results of the N-Queens Problem for $5 \leq N \leq 20$

Several observations can be obtained from Figure 5.1. First, ST1 is always the slowest among the four. This is due to the fact that standard backtracking is always inferior to forward checking for solving this problem. Second, ST2 is faster than ST1. This is due to the speed-up obtained from using a Prolog compiler. Third, FC and FCF outperform ST1 and ST2. This clearly indicates that forward checking, as used in FC and FCF, is a superior strategy than standard backtracking. Moreover, for large N (i.e. $N \geq 14$), there is a further speed-up when the first fail principle is used. The possible explanation is that in this range of N, the first fail principle helps to further eliminate part of the search space by instantiating first those more constrained d-variables.

The statistics of FC and FCF are compared with that of CHIP in the following table:

N	CC	FC	CCF	FCF
6	0.45	0.23	0.53	0.25
8	1.42	0.78	1.76	0.78
10	1.90	0.9	1.25	0.57
12	4.40	2.25	4.73	2.22
14	28.85	14.97	5.29	2.17
16	164.32	86.20	3.17	1.07

Table 5.3: Results of the N-Queen Problem

The programs CC and CCF are the equivalences of FC and FCF. FC and FCF are always twice as fast as CC and CCF. The difference may be accounted for by the following reason. WUP3.F does not implement finite domains of constants. CHIP may require some computation to distinguish different types of domains when built-in predicates are used.

5.2.2. A Logic Puzzle

This example is to solve the famous puzzle "SEND + MORE = MONEY". The purpose of the puzzle is to assign each letter a distinct digit such that the equation can be satisfied. Linear equations are used as active constraints in this example. Since WUP3.1 and SICSTUS cannot handle linear equations, only WUP3.F and CLP(R) are used to solve the puzzle.

The puzzle involves two types of constraints. The first type of constraints ensures that each letter represents a distinct digit. The **alldifferent** predicate can be used for this purpose. It defines an inequality constraint between each pair of letters.

The second type of constraints solves the equation "SEND + MORE = MONEY". It is not difficult to see that both S and M cannot be zero. Two methods can be used to represent the equation as constraints. First, the equation can be defined as follows:

$$R1 = M$$
$$R2 + S + M = O + 10 \times R1$$
$$R3 + E + O = N + 10 \times R2$$
$$R4 + N + R = E + 10 \times R3$$
$$D + E = Y + 10 \times R4$$

R1, R2, R3, and R4 are carries involved in the calculation. Each of them must be either 0 or 1 except that R1 must be 1. Second, a more natural expression can be used:

$$1000 * S + 100 * E + 10 * N + D$$
$$+ 1000 * M + 100 * O + 10 * R + E$$

$$= 10000 * M + 1000 * O + 100 * N + 10 * E + Y$$

Four programs are written: W1 is written in WUP3.F using the first method; W2 is written in WUP3.F using the second method; C1 and C2 are written in CLP(R) in a similar fashion. W1 and W2 use forward checking and the first fail principle in WUP3.F whereas C1 and C2 must be the standard backtracking approach. Yet CLP(R) allows coroutining: it will delay any unsolved constraint until more is known about the constraint.

To make the comparison complete, E1 and E2, two programs written in CHIP, are included in the following table which shows the results of all six programs:

C1	W1	E1	C2	W2	E2
9.57	.37	.30	500.66	.33	.25

Table 5.4: Results of the "SEND + MORE = MONEY" Puzzle

Table 5.4 shows that the execution times for E1, E2, W1, and W2 are quite close whereas the execution times for C1 and C2 are much higher. Since CLP(R) uses a built-in constraint solver, the efficiency of the solver increases as the number of constraints increases. This is the reason why C1 is much faster than C2.

5.2.3. The Mastermind Game

This example illustrates the ability of using constraints as choices. The problem itself is used as an example to see how the efficiency of Prolog can be improved [van88]. Basically, the purpose of the game is to guess a secret code which is a 4-digit number. Each digit is distinct. Based on the source code of this example given in [van88], four programs are written for this example:

Program	System	Approach	Time
WM1	WUP3.1	generate and test	1215.36
ST1	SICSTUS	standard backtracking + coroutining	26.33
WM2	WUP3.F	forward checking	0.27
WM3	WUP3.F	forward checking + first fail principle	0.35

Table 5.5: Results of the Mastermind Game

The first thing observed from Table 5.5 is that WM1 has the largest value in its execution time. This is due to the use of the generate and test approach. A 4-digit number must be generated before it can be tested whether it is consistent with the

previous guesses. Secondly, for ST1, standard backtracking can be utilized by using the dif(X,Y) predicate which tests the inequality of X and Y. The dif(X,Y) predicate is a form of coroutining mechanism because if any of its arguments is not ground, the predicate will be delayed until both of them are ground. Even implemented in SICSTUS, ST1 has a longer execution time than those of WM2 and WM3. Thirdly, first fail principle does not help as the execution of WM3 is larger than WM2. In other words, the overhead involved in using the first fail principle is more than the speed-up gained. A similar program written in CHIP requires 0.7 sec. [van88].

5.2.4. Map Coloring

This example is to solve an instance of the well-known map coloring problem. It concerns the assignment of the least number of colors to the vertices of a given graph so that no two adjacent vertices get the same color. It also illustrates how the branch and bound techniques are used inside WUP3.F. The minimize_maximum predicate is used to solve the problem. It minimizes the maximum number of colors that are used in the coloring process. The program is basically given as follows [van87]:

```
color_graph(Res) ←
  state_constraint(Res) &
  minimize_maximum(labelling(Res),Res);
```

where state_constraint(Res) is used to create a list of d-variables that represent the vertices. It also states the inequality constraints between two adjacent vertices while labelling(Res) serves as a generator of values for the d-variables. A program WC1 is written in WUP3.F using the above representation to solve the coloring problem for a graph that has 110 vertices and 318 edges [Gar75]. Besides, SC1 is written in SICSTUS Prolog to solve the same problem. Since the branch and bound techniques cannot be used in SICSTUS, the following program is written:

```
color_graph(Res) ←
   state_constraint(Res) &
   labelling(Res);

state_constraint(Res) )<-
   dif(R1,R2) & ... & dif(RI,RJ) & ... &
   labelling([R1,....., RN]);

labelling([]);
labelling([X|Y]) ←
      member(X,[1,2,3,4]) &
      labelling(Y);
```

Notice that the above program already narrows down the possible number of colors. EC1 denotes the equivalent program that is written in CHIP. The results are shown as follows:

Name	Ts	To	Tp	Tt
WC1	14.58	.65	1.60	16.83
EC1	0.37	2.67	2.5	5.55
SC1	N/A	N/A	N/A	68.67

Table 5.6: Results for the Map Coloring Problem

In Table 5.6, Ts is the set-up time for the problem, To is the time to give the first appearance of the optimal solution, Tp is the time between the first appearance of the optimal solution and the end of the proof (a branch and bound search has to complete the search to verify the optimal solution), and Tt is the total time. Several observations can be obtained from the table. First, since SC1 does not use the branch and bound technique, Ts, To and Tp are irrelevant. Second, Ts for WC1 is large. This is due to the restriction in WUP: a clause can have at most thirty variables. Some extra programming is needed to circumvent the restriction. WC1 is better than EC1 in To and Tp. Third, the total execution time of WC1 (i.e. 2.25) is better than that of SC1 which is a

compiled program with coroutining using the dif predicate.

5.2.5. A Scheduling Problem

This example is to show that WUP3.F can be used to solve a real-life scheduling problem. The description of the problem is given in [van87]. The problem concerns the scheduling of a construction project that erects a five-segment bridge. The project consists of a set of 46 tasks. There are several constraints between these tasks. For example, there are six excavations which require the use of a excavator. Since there is only one excavator, it is not possible for two excavation tasks to proceed simultaneously. The major feature of this problem is the use of disjunctive constraints as choice points; it also shows the representation required for this kind of constraints. A program called WS1 is written in WUP3.F to solve this problem. The execution time of WS1 is as follows. WS1 gets the first solution 110 at 11.38 seconds. The optimal solution is obtained at 26.55 seconds. The proof of optimality finishes at 281.33 seconds (i.e. 4.7 minutes). A similar program written in CHIP gets the first solution at 20.0 seconds. It obtains the optimal solution at about 1 minute. The proof of optimality requires 6 minutes. With redundant constraints [van87], the optimal solution and its proof finish at about 4.5 minutes.

5.3. Summary

This chapter has shown the performance analysis of WUP3.F using five examples. WUP3.F is able to provide the best approach to solve these examples. In several examples, it outperforms WUP3.1, CLP(R) and SICSTUS. The execution times of programs written in WUP3.F are either very close to or smaller than that of programs written in CHIP. In addition, for large problems, such as the N-queens problem for $N \geq 14$,

the speed-up gained through the use of consistency techniques is significant.

Chapter Six
Conclusion

In this chapter, the results of this research are summarized. Several future research directions are discussed. Moreover, some of the latest developments in constraint logic programming are explained briefly.

6.1. Summaries of Results

The major result of this research is that it is possible to embed consistency techniques in a conventional Prolog system. WUP3.F is an implementation of CTs in WUP3.1. This thesis has shown the necessary modifications in a Prolog interpreter to put the CTs in use.

Moreover, the requirements of an appropriate goal delay mechanism for a conventional Prolog system that can handle the arbitrary delays and activations of subgoals are identified. The model proposed by Boizumault is modified to implement the delay mechanism. One of the major requirements is to check whether a selected subgoal meets its precondition for execution or not, same as in the wait declaration in Mu-Prolog. All the necessary data structures used by the delay mechanism are explained.

Last but not least, most of the built-in predicates have been successfully implemented. Two of the more significant categories are the branch and bound technique and the handling of linear equations and inequations. All these predicates provide an efficient means to solve a constraint and to prune the search space as soon as possible.

6.2. Further Directions

The main concern of this thesis is to provide a logic programming language that can handle CSPs effectively. Since compilation reduces the execution times of logic programs, a compiler for logic programs that use CTs should be investigated. One of the difficult problems is the management of finite domains[van87].

On the other hand, it is very difficult to debug programs written in WUP3.F. The problem is due to the fact that a goal can be delayed and activated later according to the bindings of variables. Building a debugger that can monitor program execution should be considered. The management of activated goal presents a difficult problem. The debugger must have some knowledge of the delay mechanism in order to keep track of the control flow of the program.

6.3. New Developments in Constraint Logic Programming

6.3.1. CAL

Constrainte Avec Logique (CAL) [ASS88] has been developed by Institution for New Generation Computer Technology, Tokyo, Japan (ICOT). It is actually a family of languages where each language is itself an instance of CLP. The semantics of CAL is similar to that of CLP except that in CAL, constraint symbols are distinct from predicate symbols. Similar to CLP, the constraint solver in CAL not only will determine the satisfiability of a set of constraints but also will compute its canonical form.

Currently, CAL has been implemented on DEC 2060 and PSI. A pre-processor is used to translate CAL source program into an equivalent Prolog program which replaces each constraint with a special built-in predicate that invokes the constraint solver explicitly. At this moment, there are three different CALs implemented at ICOT.

The first one is the Algebraic CAL whose algorithm for its constraint solver is based on the Buchberger algorithm that computes Gröbner bases of polynomials. It is used in computer algebra and geometrical theorem proving.

The second one is the Boolean CAL. Its constraint solver is based on the approach of Boolean Gröbner-bases. Its computation domain is the set of truth-values. The third one is the Linear CAL in which linear equations and inequations can be used as constraints. A simplex-based algorithm is under development as the core of its constraint solver.

A further attempt to combine these three systems together is under way. Typed CAL is proposed so that users can use multiple constraint solvers simultaneously in a single instance of CAL. Each parameter is typed so that during the execution of a program, a suitable constraint solver can be selected according to the type of each constraint. Moreover, a new constraint solver that deals with real closed field is being developed. The ultimate goal of the current research in CAL is the parallelization of both the inference engine and the constraint solver that may lead to the design of a parallel CLP language.

6.3.2. CHIP

In a recent overview [DvS88], a new computation domain is added into CHIP. The language can now handle systems of linear equations and inequations as constraints. The constraint solver uses a simplex algorithm that is based on variable elimination. This particular constraint solver is very similar to that of CLP(R) in that it does not only determine the satisfiability of a set of constraints but also computes its solving form.

Moreover, there are two new constructs developed in CHIP. The first one is the demon declaration. Predicates that are subjected to a demon declaration are used as rewriting rules. However, they are deterministic in that a goal can only match with the head of its set of definition clauses. The other construct is the if_then_else construct which is an extension of the if_then_else of Mu-Prolog. It is of the following form:

if CONDITION then GOAL-1 else GOAL-2

A special procedure is used to determine the truth value of all instances of CONDITION. If it can be determined, either GOAL-1 or GOAL-2 will be proved according to the truth value. Otherwise, the entire construct is delayed to wait for more information.

6.3.3. Others

There are two systems — BNR Prolog and Trilogy — that represent other directions in approaching constraint logic programming. BNR Prolog [OVF88] supports relational arithmetic operations on closed intervals. All arithmetic terms are defined as closed intervals. Constraints can be expressed as arithmetic operations between these terms. Local propagation is the main technique used to solve constraints. The idea is to narrow the intervals of all the involved variables. Since arithmetic operations are relational, inversion functions are defined implicitly. Since local propagation is a weak method, it is inadequate to handle large problems that involve lots of constraints.

Trilogy [Vod88] is a constraint based logic programming language. It is based on first order predicate calculus. Constraints are mapped into the decidable theories of logic. The semantics of the language is based on the "theory of pairs". The computation rule of Trilogy can be summarized as

> *Keep substituting for the predicate calls while transforming the right hand side formula to the disjunctive normal form, performing at the same time simplifications on the constraints and eliminating the*

existential quantifiers [Vod88].

The language has been fully implemented as a native-code compiler for the IBM-PCs.

References

[ASS88] Akira Aiba , Ko Sakai, Yosuke Sato and David J. Hawley, Constraint Logic Programming Language CAL, *Proc. of the International Conference on Fifth Generation Computer Systems*, 1988, 263-276.

[Boi86] P. Boizumault, A General Model to Implement Dif and Freeze, *Proc. of the 8th International Conf. on Logic Programming*, 1986, 585-592.

[Bru82] M. Bruynooghe, The Memory Management of Prolog Implementations, in *Logic Programming, A.P.I.C. Studies in Data Processing*, K. L. Clark and S. A. Tarnlund (ed.), Academic Press, New York, 1982, 83-98.

[Car88] Mats Carlsson, Freeze, Indexing, and Other Implementation Issues in the WAM, *Proc. of the Fourth International Logic Programming Conf.*, 1988, 40-58.

[CaW88] Mats Carlsson and Johan Widen, *SISCtus Prolog User's Manual*, Research Report SICS R88007, Swedish Institute of Computer Science, 1988.

[Che84] Mantis Cheng, *Design and Implementation of the Waterloo Unix Prolog Environment*, M. Math. Thesis, University of Waterloo, Waterloo, Ontario, Canada, 1984.

[ClM80] K. L. Clark and F. G. McCabe, *IC-PROLOG: aspects of its implementation*, Proc. of International Workshop on Logic Programming, von Neumann Computer Science Society, Debrecen, Hungary, July 1980.

[Col82] Alain Colmerauer, Prolog and Infinite Trees, in *Logic Programming, A.P.I.C. Studies in Data Processing*, K. L. Clark and S. A. Tarnlund (ed.), Academic Press, New York, 1982.

[Col87] Alain Colmerauer, Opening the Prolog III Universe, *Byte*, Aug. 1987, 177-182.

[DSv87] Mehmet Dincbas, Helmut Simonis and Pascal van Hentenryck, Extending Equation Solving and Constraint Handling in Logic Programming, Tech. Rep.-LP-2203, ECRC, Feb. 1987.

[DvS88] M. Dincbas, Pascal van Hentenryck, H. Simmonis, A. Aggoun, T. Graf and F. Berthier, The Constraint Logic Programming Lauguage CHIP, *Proc. of the International Conference on Fifth Generation Computer Systems*, 1988, 693-702.

[DSv88] Mehmet Dincbas, Helmut Simonis and Pascal van Hentenryck, *Solving a Cutting-Stock Problem in Constraint Logic Programming* , Proceedings of Symposium on Logic Programming, 1988.

[Fre78] E. C. Freuder, Synthesizing Constraint Expressions, *Comm. ACM 21*, (Nov. 1978), 958-966.

[Gal87] H. Gallaire, Boosting Logic Programming, *Proc. of International Conf. on Logic Programming*, 1987, 963-988.

[Gar75] M. Gardner, *Science American*, Apr. 1975, 125.

[HJM87] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, Peter Stuckey and Ronald Yap, *The CLP(R) Programmer's Manual(version 2.02)*, Dept. of Computer Science, Monash University, Victoria, Australia, Oct. 1987.

[Hog84] C. J. Hogger, *Introduction to Logic Programming*, Academic Press, Toronto, 1984.

[JLM86] J. Jaffer, J. L. Lassez and M. J. Maher, A Logic Programming Language Scheme, in *Functional and Logic Programming*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986.

[JaL87] J. Jaffer and J. L. Lassez, *Constraint Logic Programming*, Proc. of the Conf. of Principles of Programming Languages, Munich, Jan. 1987.

[Knu75] Knuth, *Backtracking*, Mathematics of Computation, 1975.

[Kow74] R. A. Kowalski, Predicate Logic as a Programming Language, *Proc. IFIP Cong.*, 1974, 569-574.

[Las87] Catherine Lassez, Constraint Logic Programming, *Byte*, Aug. 1987, 171-176.

[Lau78] Jean-Louis Lauriere, A Language and a Program for Stating and Solving Combinatorial Problems, *Artificial Intelligence 10*, (1978), 29-127.

[Lel88] Wm Leler, *Constraint Programming Languages: Their Specification and Generation*, Addison Wesley, Don Mills, Ontario, 1988.

[Llo84] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlay, New York, 1984.

[Mac77] Alan K. Mackworth, Consistency in Networks of Relations, *Artificial Intelligence 8*, (1977), 99-118.

[Mel82] C. S. Mellish, An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter, in *Logic Programming, A.P.I.C. Studies in Data Processing*, K. L. Clark and S. A. Tarnlund (ed.), Academic Press, New York, 1982, 99-106.

[Nai85] Lee Naish, *Negation and Control in Prolog*, PhD Thesis, Univ. of Melbourne, Australia, 1985.

[OVF88] William Older, Andre Vellino and Bijan Farrahi, *Relational Arithmetic in Prolog using Interval Constraints*, Computing Research Laboratory, Bell-Northern Research, 1988.

[Rob77] G. M. Roberts, *An implementation of PROLOG*, M.Sc. Thesis, Univ. of Waterloo, Ontario, Canada, 1977.

[Rob65] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *J. ACM 15*, 4 (1965), 620-646.

[vaK76] M. H. van Emden and R. A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *J. ACM 23*, 4 (Oct. 1976), 733-742.

[van82] M. H. van Emden, An interpreting algorithm for Prolog programs, *Proc. of the First International Logic Programming Conf.*, 1982.

[vaD86] Pascal van Hentenryck and Mehmet Dincbas, Domains in Logic Programming, *IJCAI 86*, 1986, 759-765.

[van87] Pascal van Hentenryck, *Consistency Techniques in Logic Programming*, PhD Thesis, Facultes Universitaires Notre-Dame de la Paix Namur-Belgium, 1987.

[van88] Pascal van Hentenryck, A Constraint Approach to Mastermind in Logic Programming, *SIGART Newsletter*, 1988, 31-35.

[Vod88] Paul J. Voda, *The Constraint Language Trilogy: Semantics and Computations*, Complete Logic Systems Inc., Vancouver, B.C., Jan. 1988.

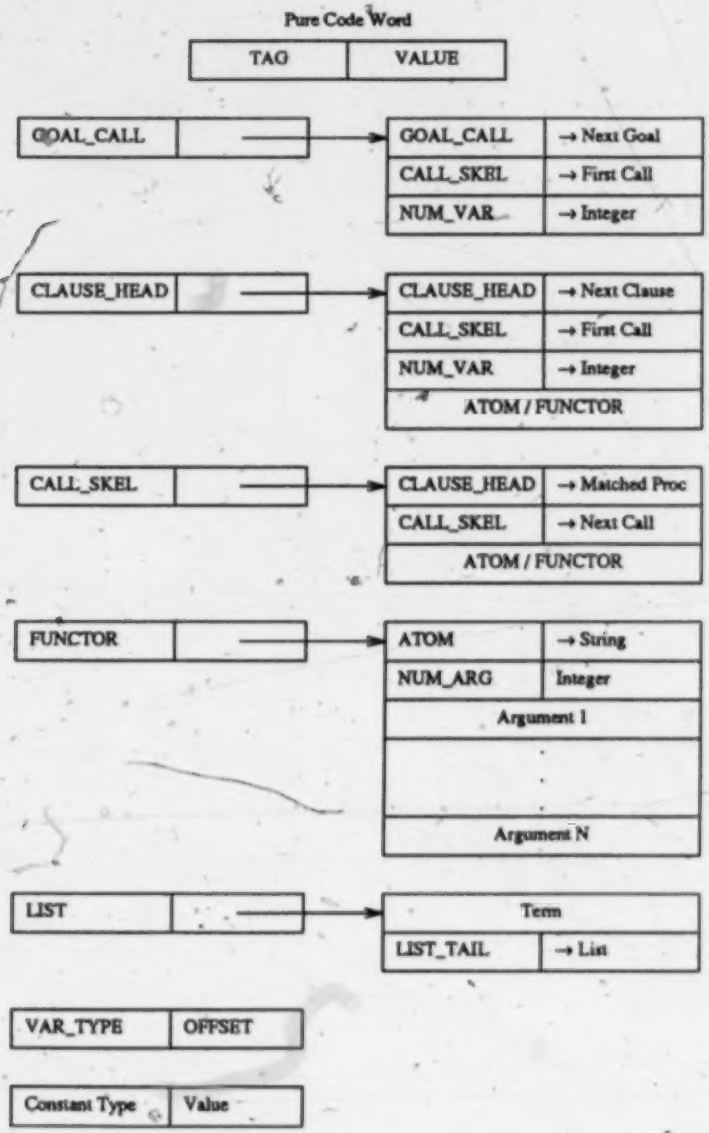
[War77] D. H. D. Warren, *Implementing PROLOG - compiling predicate logic programs*, Research Reports Nos. 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, Scotland, 1977.

[Wol86] D. A. Wolfram, *Intractable Unifiability problems and backtracking*, Third International Conference on Logic Programming, London, UK, July 1986.

[Won88] Brian Wong, *Intelligence Backtracking in Prolog*, M. Sc. Thesis, Dept. of Computer Science, Univ. of Alberta, Edmonton, Canada, Dec. 1988.

Appendix A1

Pure Code Representation of WUP



Appendix A2

Unification Table of WUP

CALL / HEAD	Free Var.	Void Var.	Int. Const	Atom Const	Char Const	End List	Const List	Var. List	Const Func	Var. Func
Free Variable	FV	S	AH	AH	AH	AH	AH	CH	AH	CH
Void Variable	S	S	S	S	S	S	S	S	S	S
Integer Constant	AC	S	SC	F	F	F	F	F	F	F
Atom Constant	AC	S	F	SC	F	F	F	F	F	F
Character Constant	AC	S	F	F	SC	F	F	F	F	F
End List Character	AC	S	F	F	F	SC	F	F	F	F
Constant List	AC	S	F	F	F	F	UL	UL	F	F
Variable List	CC	S	F	F	F	F	UL	UL	F	F
Constant Functor	AC	S	F	F	F	F	F	F	UF	UF
Variable Functor	CC	S	F	F	F	F	F	F	UF	UF

F	-	always fail	UL	-	unify lists
S	-	always succeed	UF	-	unify functors
FV	-	free variables	CH	-	copy to head
AH	-	assign to head	CC	-	copy to call
AC	-	assign to call	SC	-	simple comparison

Appendix A3

An Algorithm for the Reduce Predicate

This algorithm is simplified from the one given in [Lau78]:

1) Express the constraint as $T^+ + T^- \; © \; 0$ where T^+ is the set of positive terms in the constraint, T^- is the set of negative terms and © is the connective.

2) Calculate the intervals [P,PP], [N, NN] and [A,B] where

$$P = \sum_{i \in T^+} \min t_i, \quad PP = \sum_{i \in T^+} \max t_i,$$

$$N = \sum_{i \in T^-} \min t_i, \quad NN = \sum_{i \in T^-} \max t_i, \text{ and}$$

$$[A,B] = [P,PP] \cap [N,NN]$$

3) Depending on the values of the intervals, the following cases are possible:

i) if A > B
  if A = P & © is ≥
    the constraint is trivially true
  else
    the constraint is never satisfied

ii) if A = B
  if A = P = NN & © is ≥
    the constraint is trivially true
  else
    two extra constraints can be observed:

$$\sum_{i \in T^+} t_i = A \quad \text{and} \quad \sum_{i \in T^-} t_i = A$$

iii) if A < B

  if P < N

$$\sum_{i \in T^+} t_i \geq A \text{ is inferred as a new constraint}$$

  else if P > N © is =

$$\sum_{i \in T^-} t_i \geq A \text{ is inferred as a new constraint}$$

  if PP > NN & © is =

$\sum_{i \in T^-} t_i \leq B$ is inferred as a new constraint

else if PP > NN

$\sum_{i \in T^-} t_i \leq B$ is inferred as a new constraint

4) Process the new constraints obtained in step 3). Each new constraint has a special property: either $|T^+| = 1$ or $|T^-| = 1$. For a constraint with $|T^-| = 1$, the following constraint can be applied to reduce the domain of the term t_{i_0}:

$$t_{i_0} \geq NN - \sum_{i \in T^-, i \neq i_0} \max t_i$$

and if © is =, a further constraint can be applied:

$$t_{i_0} \leq N - \sum_{i \in T^-, i \neq i_0} \min t_i$$

For a constraint with $|T^+| = 1$, the following constraint can be applied to reduce the domain of the term t_{i_0}:

$$t_{i_0} \geq PP - \sum_{i \in T^-, i \neq i_0} \max t_i$$

and if © is =, a further constraint can be applied:

$$t_{i_0} \leq P - \sum_{i \in T^-, i \neq i_0} \min t_i$$

Appendix A4

Program Listings of Examples in Chapter Five

The N-queens Problem

1) ST1 - written in WUP3.1 using the standard backtracking approach. ST1 is based on the code described in [van87].

```
solution([],Q,Q);
solution(L,TL,Q) <-
      del(A,L,L1) &
      append(TL,[A],TL1) &
      safe(TL1) &
      solution(L1,TL1,Q);

del(A,[A|L],L);
del(A,[B|L],[B|L1]) <-
      del(A,L,L1);

safe([]);
safe([Q|O]) <-
      safe(O) &
      noattack(Q,O,1);

noattack(_,[],_);
noattack(Y,[Y1|L],D) <-
      T1 is Y - Y1 &
      T1 <> D &
      T2 is Y1 - Y &
      T2 <> D &
      D1 is D + 1 &
      noattack(Y,L,D1);
```

2) ST2 - is written in SICSTUS using the standard backtracking approach. ST2 is modified from ST1.

```
solution([],Q,Q).
solution(L,TL,Q) :-
      del(A,L,L1),
      append(TL,[A],TL1),
      safe(TL1),
      solution(L1,TL1,Q).

del(A,[A|L],L).
del(A,[B|L],[B|L1]) :-
      del(A,L,L1).

safe([]).
safe([Q|O]) :-
      safe(O),
      noattack(Q,O,1).
```

```
noattack(_,[],_).
noattack(Y,[Y1|L],D) :-
        T1 is Y - Y1,
        T1 == D,
        T2 is Y1 - Y,
        T2 == D,
        D1 is D + 1,
        noattack(Y,L,D1).

append([],X,X).
append([X|Y],Z,[X|L]) :-
        append(Y,Z,L).
```

3) FC - written in WUP3.F using the forward approach. FC is based on the code described in [van87].

```
@Dqueen(X) <-
        safe(X) &
        labelling(X);

safe([]);
safe([F|T]) <-
        noattack(F,T) &
        safe(T);

labelling([]);
labelling([X|Y]) <-
        indomain(X) &
        labelling(Y);

noattack(X,Y) <-
        noattack(X,Y,1);

noattack(X,[],Nb);
noattack(X,[Y|YT],Nb) <-
        Y !=@ X &
        X <>@ Y + Nb &
        X <>@ Y - Nb &
        Nb1 is Nb + 1 &
        noattack(X,YT,Nb1);

/* for a 5-queens problem */

?domain @Dqueen(1 .. 5);
?eq(X,[X1,X2,X3,X4,X5]) & @Dqueen(X) & cut;
```

4) FCF - it is written in WUP3.F using the forward approach and the first fail principle. The program is similar to FC except that the definition of the labelling predicate is different:

```
labelling([]);
labelling([X|Y]) <-
        deleteff(V,[X|Y],L1) &
```

```
    indomain(V) &
    labelling(Ls);
```

The "SEND + MORE = MONEY" Puzzle

1) C1 - written in CLP(R) using the standard backtracking approach. The equation is expressed in term of five equalities. C1 is a sample program in the CLP(R) interpreter package.

```
p(S, E, N, D, M, O, R, Y) :-
    S > 0, E >= 0, N >= 0, D >= 0, M > 0, O >= 0, R >= 0, Y >= 0,
    S <= 9, E <= 9, N <= 9, D <= 9, M <= 9, O <= 9, R <= 9, Y <= 9,
    D + E = Y + 10*C1,
    C1 + N + R = E + 10*C2,
    C2 + E + O = N + 10*C3,
    C3 + S + M = O + 10*M,
    carry(C1, C2, C3),
    dig(S), dig(E), dig(N), dig(D), dig(M), dig(O), dig(R),
    dig(Y),
    difflist([S, E, N, D, M, O, R, Y]).

carry(1, 1, 1). carry(1, 1, 0). carry(1, 0, 1). carry(1, 0, 0).
carry(0, 1, 1). carry(0, 1, 0). carry(0, 0, 1). carry(0, 0, 0).

del(X,[X|L],L).
del(X,[Y|L],L1) :-
    del(X,L,L1).

dig(9). dig(8). dig(7). dig(6). dig(5).
dig(4). dig(3). dig(2). dig(1). dig(0).

difflist([X | T]) :-
    notmem(X, T),
    difflist(T).
difflist([]).

notmem(X, [Y | Z]) :-
    X < Y,
    notmem(X, Z).
notmem(X, [Y | Z]) :-
    X > Y,
    notmem(X, Z).
notmem(X, []).

go:-
    p(S, E, N, D, M, O, R, Y),
    printf("\nAns:\n",[]),
    printf("S = %d, E = %d, N = %d, D = %d, M = %d, O = %d,
    R = %d, Y = %d \n", [S,E,N,D,M,O,R,Y]).
```

2) C2 - written in CLP(R) using the standard backtracking approach. The equation is expressed in term of one equality. C2 is similar to C1 except that the carry predicate is

not used and the definition of the predicate p is different. C2 is modified from C1.

```
p(S, E, N, D, M, O, R, Y) :-
      S > 0, E >= 0, N >= 0, D >= 0, M > 0, O >= 0, R >= 0, Y >= 0,
      S <= 9, E <= 9, N <= 9, D <= 9, M <= 9, O <= 9, R <= 9, Y <= 9,
      1000 * S + 100 * E + 10 * N + D +
      1000 * M + 100 * O + 10 * R + E =
      10000 * M + 1000 * O + 100 * N + 10 * E + Y,
      dig(S), dig(E), dig(N), dig(D), dig(M), dig(O), dig(R),
      dig(Y),
      difflist([S, E, N, D, M, O, R, Y]).
```

3) W1 - written in WUP3.F using forward checking and the first fail principle. The equation is expressed in term of five equalities. W1 is based on [van87].

```
?clear_all;
?domain @Dsendmory(0 .. 9, 0 .. 1);

test(X,Y) <-
      @Dsendmory(X,Y);

@Dsendmory([S,E,N,D,M,O,R,Y], [R1,R2,R3,R4]) <-
      alldifferent([S,E,N,D,M,O,R,Y]) &
      S !=@ 0 &
      M !=@ 0 &
      R1 = M &
      reduce(D + E = Y + 10 * R4) &
      reduce(R2 + S + M = O + 10 * R1) &
      reduce(R3 + E + O = N + 10 * R2) &
      reduce(R4 + N + R = E + 10 * R3) &
      labelling([S,E,N,D,M,O,R,Y,R1,R2,R3,R4]) &
      write([_,S,E,N,D]) & nl &
      write([_,M,O,R,E]) & nl &
      write([M,O,N,E,Y]) ;

alldifferent([]);
alldifferent([X|Y]) <-
      outof(X,Y) &
      alldifferent(Y);

outof(X,[Y|Ls]) <-
      X !=@ Y &
      outof(X,Ls);
outof(X,[]);

labelling([X|Y]) <-
      deleteff(V,[X|Y],L) &
      indomain(V) &
      labelling(L);
labelling([]);
```

4) W2 - written in WUP3.F using forward checking and the first fail principle. The equation is expressed in term of one equality. W1 is modified from W1.

```
?clear_all;
?domain @Dsendmory(0 .. 9);

test(X) <-
      @Dsendmory(X);

@Dsendmory([D,E,M,N,O,R,S,Y]) <-
      alldifferent([D,E,M,N,O,R,S,Y]) &
      S !=@ 0 &
      M !=@ 0 &
      reduce(1000 * S + 100 * E + 10 * N + D + 1000 * M + 100 * O +
            10 * R + E =
      10000 * M + 1000 * O + 100 * N + 10 * E + Y) &
      labelling([D,E,M,N,O,R,S,Y]) &
      write([_,S,E,N,D]) & nl &
      write([_,M,O,R,E]) & nl &
      write([M,O,N,E,Y]) & nl;

alldifferent([]);
alldifferent([X|Y]) <-
      outof(X,Y) &
      alldifferent(Y);

outof(X,[]);
outof(X,[Y|Ls]) <-
      X !=@ Y &
      outof(X,Ls);

labelling([]);
labelling([X|Y]) <-
      deleteff(V,[X|Y],L) &
      indomain(V) &
      labelling(L);
```

The Mastermind Game

1) ST1 - written in SICSTUS using the standard backtracking approach with coroutining. ST1 is modified from WM2.

```
mastermind(Code,Prev) :-
      guess_code(Guess,Prev) ,
      ask_the_user(Guess, Score) ,
      mastermind_aux(Code,Prev,Score).

mastermind_aux(Code,Prev,[Code,4,_]).
mastermind_aux(Code,Prev,[Guess,Bulls,Cows]) :-
      Bulls == 4 ,
      mastermind(Code,[[Guess,Bulls,Cows]|Prev]).

guess_code([X1,X2,X3,X4],Prev) :-
      alldifferent([X1,X2,X3,X4]) ,
      consistent([X1,X2,X3,X4],Prev) ,
```

```
        labelling([X1,X2,X3,X4]).

labelling([]).
labelling([X|Y]) :-
        member(X,[0,1,2,3,4,5,6,7,8,9]),
        labelling(Y).
alldifferent([]).
alldifferent([X|Ls]) :-
        outof(X,Ls) ,
        alldifferent(Ls).

outof(X,[]).
outof(X,[Y|Ls]) :-
        dif(X,Y),
        outof(X,Ls).

consistent(Guess, []).
consistent(Guess,Prev) :-
        consistent_bulls(Guess,Prev) ,
        consistent_cows(Guess,Prev).

consistent_bulls(Guess,[]).
consistent_bulls(Guess,[[Prev,Bulls,Cows]|Ps]) :-
        exact_matches(Guess,Prev,Bulls) ,
        consistent_bulls(Guess,Ps).

consistent_cows(Guess,[]).
consistent_cows(Guess,[[Prev,Bulls,Cows]|Ps]) :-
        BullsCows is Bulls + Cows ,
        common_members(Guess,Prev,BullsCows) ,
        consistent_cows(Guess,Ps).

exact_matches(Xs,Ys,N) :-
        exact_matches(Xs,Ys,0,N).

exact_matches([],[],N,N).
exact_matches([X|Xs],[X|Ys],K,N) :-
        K1 is K + 1 ,
        exact_matches(Xs,Ys,K1,N).
exact_matches([X|Xs],[Y|Ys],K,N) :-
        dif(X,Y) ,
        exact_matches(Xs,Ys,K,N).

common_members(Xs,Ys,N) :-
        common_members_g(Xs,Ys,0,N).

common_members_g([X|Xs],Ys,K,N) :-
        member(X,Ys) ,
        K1 is K + 1 ,
        common_members_g(Xs,Ys,K1,N).
```

```
common_members_g([X|Xs],Ys,K,N) :-
      outof(X,Ys) ,
      common_members_g(Xs,Ys,K,N).

common_members_g([],Ys,N,N).

ask_the_user(Guess,[Guess,Bulls,Cows]) :-
      write('Guess is ') , write(Guess) , nl ,
      write('Bulls = ?') ,
      read(Bulls) ,
      nl ,
      write('Cows = ?') ,
      read(Cows) ,
      nl ,
      ! ,
      write([Guess,Bulls,Cows]) , nl.

append([X|Y],Z,[X|L]) :-
      append(Y,Z,L).

member(X,[X|_]).
member(X,[Y|L]) :-
      member(X,L).
```

2) WM1 - written in WUP3.1 using the generate and test approach. WM1 is modified from WM2.

```
?clear_all;

mastermind(Code,Prev) <-
      guess_code(Guess,Prev) &
      cut &
      ask_the_user(Guess, Score) &
      mastermind_aux(Code,Prev,Score);

mastermind_aux(Code,Prev,[Code,4,_]);
mastermind_aux(Code,Prev,[Guess,Bulls,Cows]) <-
      Bulls <> 4 &
      mastermind(Code,[[Guess,Bulls,Cows]|Prev]);

guess_code(Guess,Prev) <-
      generatelist(Guess) &
      consistent(Guess,Prev) ;


generatelist(G) <-
      generatelist([X1,X2,X3,X4], [], G);
generatelist([], G, G);
generatelist([X|Xs],L, G ) <-
      member(X,[0,1,2,3,4,5,6,7,8,9]) &
      outof(X,L) &
      append(L,[X],NL) &
```

```
        generatelist(Xs,NL, G);

consistent(Guess, []);
consistent(Guess,Prev) <-
        consistent_bulls(Guess,Prev) &
        consistent_cows(Guess,Prev);

consistent_bulls(Guess,[]);
consistent_bulls(Guess,[[Prev,Bulls,Cows]|Ps]) <-
        exact_matches(Guess,Prev,Bulls) &
        consistent_bulls(Guess,Ps);

consistent_cows(Guess,[]);
consistent_cows(Guess,[[Prev,Bulls,Cows]|Ps]) <-
        BullsCows is Bulls + Cows &
        common_members(Guess,Prev,BullsCows) &
        consistent_cows(Guess,Ps);

exact_matches(Xs,Ys,N) <-
        exact_matches(Xs,Ys,0,N);

exact_matches([],[],N,N);
exact_matches([X|Xs],[X|Ys],K,N) <-
        cut &
        K1 is K + 1 &
        exact_matches(Xs,Ys,K1,N);
exact_matches([X|Xs],[Y|Ys],K,N) <-
        X <> Y &
        exact_matches(Xs,Ys,K,N);

common_members(Xs,Ys,N) <-
        common_members_g(Xs,Ys,0,N);

common_members_g([X|Xs],Ys,K,N) <-
        member(X,Ys) &
        cut &
        K1 is K + 1 &
        common_members_g(Xs,Ys,K1,N);

common_members_g([X|Xs],Ys,K,N) <-
        outof(X,Ys) &
        common_members_g(Xs,Ys,K,N);

common_members_g([],Ys,N,N);

ask_the_user(Guess,[Guess,Bulls,Cows]) <-
        write("Guess is ") & write(Guess) & nl &
        write("Bulls = ?") &
        read_num(Bulls) &
        nl &
        write("Cows = ?") &
```

```
        read_num(Cows) &
        nl &
        cut &
        write([Guess,Bulls,Cows]) & nl;
read_num(N) <-
        read_term(N) &
        cut &
        is_int(N) ;

outof(X,[]);
outof(X,[Y|Ls]) <-
        member(X,[0,1,2,3,4,5,6,7,8,9]) &
        member(Y,[0,1,2,3,4,5,6,7,8,9]) &
        X <> Y &
        outof(X,Ls);
```

3) WM2 - written in WUP3.F using the forward checking approach. WM2 is based on [van87].

```
?clear_all;
?domain @Dalldistinct(0 .. 9);

mastermind(Code,Prev) <-
        guess_code(Guess,Prev) &
        ask_the_user(Guess, Score) &
        mastermind_aux(Code,Prev,Score);

mastermind_aux(Code,Prev,[Guess,Bulls,Cows]) <-
        Bulls <> 4 &
        mastermind(Code,[[Guess,Bulls,Cows]|Prev]);
mastermind_aux(Code,Prev,[Code,4,_]);

guess_code(Guess,Prev) <-
        @Dalldistinct(Guess) &
        consistent(Guess, Prev) &
        labelling(Guess);

@Dalldistinct([X1,X2,X3,X4]) <-
        alldifferent([X1,X2,X3,X4]);

alldifferent([X|Ls]) <-
        outof(X,Ls) &
        alldifferent(Ls);
alldifferent([]);

outof(X,[Y|Ls]) <-
        X !=@ Y &
        outof(X,Ls);
outof(X,[]);

consistent(Guess,Prev) <-
        consistent_bulls(Guess,Prev) &
```

```
        consistent_cows(Guess,Prev);
consistent(Guess, []);

consistent_bulls(Guess,[[Prev,Bulls,Cows]|Ps]) <-
        exact_matches(Guess,Prev,Bulls) &
        consistent_bulls(Guess,Ps);
consistent_bulls(Guess,[]);

consistent_cows(Guess,[[Prev,Bulls,Cows]|Ps]) <-
        BullsCows is Bulls + Cows &
        common_members(Guess,Prev,BullsCows) &
        consistent_cows(Guess,Ps);
consistent_cows(Guess,[]);

exact_matches(Xs,Ys,0) <-
        pair_dif(Xs,Ys);

exact_matches([X|Xs],[X|Ys],N) <-
        N > 0 &
        N1 is N - 1 &
        exact_matches(Xs,Ys,N1);
exact_matches([X|Xs],[Y|Ys],N) <-
        N > 0 &
        X !=@ Y &
        exact_matches(Xs,Ys,N);

pair_dif([X|Xs],[Y|Ys]) <-
        X !=@ Y &
        pair_dif(Xs,Ys);
pair_dif([],[]);

common_members(Xs,Ys,0) <-
        allnotmember(Xs,Ys) ;

common_members([X|Xs],Ys,N) <-
        N > 0 &
        member(X,Ys) &
        N1 is N - 1 &
        common_members(Xs,Ys,N1);
common_members([X|Xs],Ys,N) <-
        N > 0 &
        outof(X,Ys) &
        common_members(Xs,Ys,N);

allnotmember([X|Xs],Ys) <-
        outof(X,Ys) &
        allnotmember(Xs,Ys);
allnotmember([],Ys) ;

ask_the_user(Guess,[Guess,Bulls,Cows]) <-
        write("Guess is ") & write(Guess) & nl &
```

```
        write("Bulls = ?") &
        read_num(Bulls) &
        write("Cows = ?") &
        read_num(Cows) &
        nl;

labelling([X|Y]) <-
        indomain(X) &
        labelling(Y);
labelling([]);

read_num(N) <-
        read_term(N) &
        cut &
        is_int(N) ;
```

4) WM3 - written in WUP3.F using the forward checking approach and the first fail principle. WM3 is very similar to WM2. The only differences between the two is the definition of the labelling predicate. In WM3, it is defined as follows:

```
labelling([X|Y]) <-
        deleteff(V,[X|Y],Ls) &
        indomain(V) &
        labelling(Ls);
labelling([]);
```

The Map Coloring Problem

1) SC1 - written in SICSTUS using the standard backtracking approach and coroutining:

```
start( [R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,
R13,R14,R15,R16,R17,R18,R19,R20,R21,R22,R23,R24,
R25,R26,R27,R28,R29,R30,R31,R32,R33,R34,R35,R36,
R37,R38,R39,R40,R41,R42,R43,R44,R45,R46,R47,R48,
R49,R50,R51,R52,R53,R54,R55,R56,R57,R58,R59,R60,
R61,R62,R63,R64,R65,R66,R67,R68,R69,R70,R71,R72,
R73,R74,R75,R76,R77,R78,R79,R80,R81,R82,R83,R84,
R85,R86,R87,R88,R89,R90,R91,R92,R93,R94,R95,R96,
R97,R98,R99,R100,R101,R102,R103,R104,R105,R106,
R107,R108,R109,R110]) :-
dif(R1, R2),dif(R1, R3),dif(R1, R4),dif(R1, R5),
dif(R2, R3),dif(R2, R5),dif(R2, R6),dif(R2, R7),
dif(R2, R11), dif(R3, R4),dif(R3, R7),dif(R3, R8),
dif(R3, R9),dif(R4, R5),dif(R4, R9),dif(R4, R10),
dif(R5, R10),dif(R5, R11),dif(R6, R7),dif(R6, R11),
dif(R6, R12),dif(R6, R13),dif(R6, R19),dif(R7, R8),
dif(R7, R13),dif(R7, R14),dif(R8, R9),dif(R8, R14),
dif(R8, R15),dif(R8, R16),dif(R9, R10),dif(R9, R16),
dif(R9, R17), dif(R10, R11),dif(R10, R17),dif(R10, R18),
dif(R11, R18),dif(R11, R19),dif(R12, R13),dif(R12, R19),
dif(R12, R20),dif(R12, R21),dif(R12, R29),dif(R13, R14),
dif(R13, R21),dif(R13, R22),dif(R14, R15),dif(R14, R22),
```

dif(R14, R23),dif(R15, R16),dif(R15, R23),dif(R15, R24),
dif(R15, R25),dif(R16, R17),dif(R16, R25),dif(R16, R26),
dif(R17, R18),dif(R17, R26),dif(R17, R27),dif(R18, R19),
dif(R18, R27),dif(R18, R28), dif(R19, R28),dif(R19, R29),
dif(R20, R21),dif(R20, R29),dif(R20, R30),dif(R20, R31),
dif(R20, R41),dif(R21, R22),dif(R21, R31),dif(R21, R32),
dif(R22, R23),dif(R22, R32),dif(R22, R33),dif(R23, R24),
dif(R23, R33),dif(R23, R34),dif(R24, R25),dif(R24, R34),
dif(R24, R35),dif(R24, R36),dif(R25, R26),dif(R25, R36),
dif(R25, R37),dif(R26, R27),dif(R26, R37),dif(R26, R38),
dif(R27, R28),dif(R27, R38),dif(R27, R39),dif(R28, R29),
dif(R28, R39),dif(R28, R40),dif(R29, R40),dif(R29, R41),
dif(R30, R31),dif(R30, R41),dif(R30, R42),dif(R30, R43),
dif(R30, R55), dif(R31, R32),dif(R31, R43),dif(R31, R44),
dif(R32, R33),dif(R32, R44),dif(R32, R45),dif(R33, R34),
dif(R33, R45),dif(R33, R46),dif(R34, R35),dif(R34, R46),
dif(R34, R47), dif(R35, R36),dif(R35, R47),dif(R35, R48),
dif(R35, R49), dif(R36, R37),dif(R36, R49),dif(R36, R50),
dif(R37, R38),dif(R37, R50),dif(R37, R51),dif(R38, R39),
dif(R38, R51),dif(R38, R52),dif(R39, R40),dif(R39, R52),
dif(R39, R53), dif(R40, R41),dif(R40, R53),dif(R40, R54),
dif(R41, R54),dif(R41, R55),dif(R42, R43),dif(R42, R55),
dif(R42, R56),dif(R42, R57),dif(R42, R71),dif(R43, R44),
dif(R43, R57),dif(R43, R58),dif(R44, R45),dif(R44, R58),
dif(R44, R59), dif(R45, R46),dif(R45, R59),dif(R45, R60),
dif(R46, R47),dif(R46, R60),dif(R46, R61),dif(R47, R48),
dif(R47, R61),dif(R47, R62),dif(R48, R49),dif(R48, R62),
dif(R48, R63),dif(R48, R64),dif(R49, R50),dif(R49, R64),
dif(R49, R65),dif(R50, R51),dif(R50, R65),dif(R50, R66),
dif(R51, R52),dif(R51, R66),dif(R51, R67),dif(R52, R53),
dif(R52, R67),dif(R52, R68),dif(R53, R54),dif(R53, R68),
dif(R53, R69),dif(R54, R55),dif(R54, R69),dif(R54, R70),
dif(R55, R70),dif(R55, R71),dif(R56, R57),dif(R56, R71),
dif(R56, R72),dif(R56, R73),dif(R56, R89),dif(R57, R58),
dif(R57, R73),dif(R57, R74),dif(R58, R59),dif(R58, R74),
dif(R58, R75),dif(R59, R60),dif(R59, R75),dif(R59, R76),
dif(R60, R61),dif(R60, R76),dif(R60, R77),dif(R61, R62),
dif(R61, R77),dif(R61, R78),dif(R62, R63),dif(R62, R78),
dif(R62, R79),dif(R63, R64),dif(R63, R79),dif(R63, R80),
dif(R63, R81),dif(R64, R65),dif(R64, R81),dif(R64, R82),
dif(R65, R66),dif(R65, R82),dif(R65, R83),dif(R66, R67),
dif(R66, R83),dif(R66, R84),dif(R67, R68),dif(R67, R84),
dif(R67, R85),dif(R68, R69),dif(R68, R85),dif(R68, R86),
dif(R69, R70),dif(R69, R86),dif(R69, R87),dif(R70, R71),
dif(R70, R87),dif(R70, R88),dif(R71, R88),dif(R71, R89),
dif(R72, R73),dif(R72, R89),dif(R72, R90),dif(R72, R91),
dif(R72, R109),dif(R73, R74),dif(R73, R91),dif(R73, R92),
dif(R74, R75),dif(R74, R92),dif(R74, R93),dif(R75, R76),
dif(R75, R93),dif(R75, R94),dif(R76, R77),dif(R76, R94),
dif(R76, R95),dif(R77, R78),dif(R77, R95),dif(R77, R96),
dif(R78, R79),dif(R78, R96),dif(R78, R97),dif(R79, R80),

```
dif(R79, R97),dif(R79, R98),dif(R80, R81),dif(R80, R98),
dif(R80, R99),dif(R80, R100),dif(R81, R82),dif(R81, R100),
dif(R81, R101),dif(R82, R83),dif(R82, R101),dif(R82, R102),
dif(R83, R84),dif(R83, R102),dif(R83, R103),dif(R84, R85),
dif(R84, R103),dif(R84, R104),dif(R85, R86),dif(R85, R104),
dif(R85, R105),dif(R86, R87),dif(R86, R105),dif(R86, R106),
dif(R87, R88),dif(R87, R106),dif(R87, R107),dif(R88, R89),
dif(R88, R107),dif(R88, R108),dif(R89, R108),dif(R89, R109),
dif(R90, R91),dif(R90, R100),dif(R90, R110),dif(R91, R92),
dif(R91, R100),dif(R92, R93),dif(R92, R100),dif(R93, R94),
dif(R93, R100),dif(R94, R95),dif(R94, R100),dif(R95, R96),
dif(R95, R100),dif(R96, R97),dif(R96, R100),dif(R97, R98),
dif(R97, R100),dif(R98, R99),dif(R98, R100),dif(R99, R100),
dif(R100, R101),dif(R100, R110),dif(R101, R102),dif(R101, R110),
dif(R102, R103),dif(R102, R110),dif(R103, R104),dif(R103, R110),
dif(R104, R105),dif(R104, R110),dif(R105, R106),dif(R106, R107),
dif(R107, R108),dif(R108, R109),

labelling( [R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16,
R17,R18,R19,R20,R21,R22,R23,R24,R25,R26,R27,R28,R29,R30,R31,R32,
R33,R34,R35,R36,R37,R38,R39,R40,R41,R42,R43,R44,R45,R46,R47,R48,
R49,R50,R51,R52,R53,R54,R55,R56,R57,R58,R59,R60, R61,R62,R63,R64,
R65,R66,R67,R68,R69,R70,R71,R72,R73,R74,R75,R76,R77,R78,R79,R80,
R81,R82,R83,R84,R85,R86,R87,R88,R89,R90,R91,R92,R93,R94,R95,R96,
R97,R98,R99,R100,R101,R102,R103,R104,R105,R106,R107,R108,R109,R110]).

labelling([X|Y]) :-
      member(X,[1,2,3,4]),
      labelling(Y).
labelling([]).

member(X,[X|L].
member(X,[Y|L]) :-
      member(X,L).
```

2) WC1 - written in WUP3.F using the branch and bound method and the first fail principle. WC1 is modified from the source described in [van87].

```
?domain @Dnode(1 .. 110);
@Dnode(X);

connection( [
[r1,r2,r3,r4,r5], [r2,r6,r7,r3,r5,r11], [r3,r7,r8,r9,r4],
[r4,r9,r10,r5], [r5,r10,r11], [r6,r7,r11,r12,r13,r19],
[r7,r8,r13,r14], [r8,r9,r14,r15,r16], [r9,r10,r16,r17],
[r10,r11,r17,r18], [r11,r18,r19], [r12,r13,r19,r20,r21,r29],
[r13,r14,r21,r22], [r14,r15,r22,r23], [r15,r16,r23,r24,r25],
[r16,r17,r25,r26], [r17,r18,r26,r27], [r18,r19,r27,r28],
[r19,r28,r29], [r20,r21,r29,r30,r31,r41], [r21,r22,r31,r32],
[r22,r23,r32,r33], [r23,r24,r33,r34], [r24,r25,r34,r35,r36],
[r25,r26,r36,r37], [r26,r27,r37,r38], [r27,r28,r38,r39],
```

```
[r28,r29,r39,r40], [r29,r40,r41], [r30,r31,r41,r42,r43,r55],
[r31,r32,r43,r44], [r32,r33,r44,r45], [r33,r34,r45,r46],
[r34,r35,r46,r47], [r35,r36,r47,r48,r49], [r36,r37,r49,r50],
[r37,r38,r50,r51], [r38,r39,r51,r52], [r39,r40,r52,r53],
[r40,r41,r53,r54], [r41,r54,r55], [r42,r43,r55,r56,r57,r71],
[r43,r44,r57,r58], [r44,r45,r58,r59], [r45,r46,r59,r60],
[r46,r47,r60,r61], [r47,r48,r61,r62], [r48,r49,r62,r63,r64],
[r49,r50,r64,r65], [r50,r51,r65,r66], [r51,r52,r66,r67],
[r52,r53,r67,r68], [r53,r54,r68,r69], [r54,r55,r69,r70],
[r55,r70,r71], [r56,r57,r71,r72,r73,r89], [r57,r58,r73,r74],
[r58,r59,r74,r75], [r59,r60,r75,r76], [r60,r61,r76,r77],
[r61,r62,r77,r78], [r62,r63,r78,r79], [r63,r64,r79,r80,r81],
[r64,r65,r81,r82], [r65,r66,r82,r83], [r66,r67,r83,r84],
[r67,r68,r84,r85], [r68,r69,r85,r86], [r69,r70,r86,r87],
[r70,r71,r87,r88], [r71,r88,r89], [r72,r73,r89,r90,r91,r109],
[r73,r74,r91,r92], [r74,r75,r92,r93], [r75,r76,r93,r94],
[r76,r77,r94,r95], [r77,r78,r95,r96], [r78,r79,r96,r97],
[r79,r80,r97,r98], [r80,r81,r98,r99,r100], [r81,r82,r100,r101],
[r82,r83,r101,r102], [r83,r84,r102,r103], [r84,r85,r103,r104],
[r85,r86,r104,r105], [r86,r87,r105,r106], [r87,r88,r106,r107],
[r88,r89,r107,r108], [r89,r108,r109], [r90,r91,r100,r110],
[r91,r92,r100], [r92,r93,r100], [r93,r94,r100], [r94,r95,r100],
[r95,r96,r100], [r96,r97,r100], [r97,r98,r100], [r98,r99,r100],
[r99,r100], [r100,r101,r110], [r101,r102,r110], [r102,r103,r110],
[r103,r104,r110], [r104,r105,r110], [r105,r106], [r106,r107],
[r107,r108], [r108,r109] ]);

nodes( [r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16,
r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28,r29,r30,r31,r32,
r33,r34,r35,r36,r37,r38,r39,r40,r41,r42,r43,r44,r45,r46,r47,r48,
r49,r50,r51,r52,r53,r54,r55,r56,r57,r58,r59,r60,r61,r62,r63,r64,
r65,r66,r67,r68,r69,r70,r71,r72,r73,r74,r75,r76,r77,r78,r79,r80,
r81,r82,r83,r84,r85,r86,r87,r88,r89,r90,r91,r92,r93,r94,r95,r96,
r97,r98,r99,r100,r101,r102,r103,r104,r105,r106,r107,r108,r109,r110]);

make_data([],[]);
make_data([N|Ln],[[N,V]|Lv]) <-
      @Dnode(V) &
      make_data(Ln,Lv);

make_connection(Lv) <-
      connection(Lc) &
      make_connection(Lc,Lv);
make_connection([],Lv);
make_connection([[N|L]|Lc],Lv) <-
      member([N,V],Lv) &
      cut &
      get_connection(V,L,Lv) &
      make_connection(Lc,Lv);

get_connection(V,[],Lv);
```

```
get_connection(V,[NE|L],Lv) <-
      member([NE,VE],Lv) &
      cut &
      V !=@ VE &
      get_connection(V,L,Lv);

graph(G) <-
      nodes(Ln) &
      make_data(Ln,Lv) &
      make_connection(Lv) &
      write("graph construction completed") & nl &
      get_node(Lv,G);

get_node([[_,V]|L],[V|G]) <-
      get_node(L,G);
get_node([],[]);

labelling([X|Y]) <-
      deleteffc(V,[X|Y],L) &
      indomain(V) &
      labelling(L);
labelling([]);

start(G) <-
      graph(G) &
      stats &
      minimize_maximum(labelling(G),G);
```

The Scheduling Problem

The following program is written in WUP3.F.

```
?clear_all;
?domain @Dtask(0 .. 200);

@Dtask(X);
job([pa,a1,a2,a3,a4,a5,a6,p1,p2,ue,s1,s2,s3,s4,s5,s6,
b1,b2,b3,b4,b5,b6,ab1,ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,me,
l,t1,t2,t3,t4,t5,ua,v1,v2,k1,k2,pe]);

duration(pa,0); duration(a1,4); duration(a2,2); duration(a3,2);
duration(a4,2); duration(a5,2); duration(a6,5); duration(p1,20);
duration(p2,13); duration(ue,10); duration(s1,8); duration(s2,4);
duration(s3,4); duration(s4,4); duration(s5,4); duration(s6,10);
duration(b1,1); duration(b2,1); duration(b3,1); duration(b4,1);
duration(b5,1); duration(b6,1); duration(ab1,1); duration(ab2,1);
duration(ab3,1); duration(ab4,1); duration(ab5,1); duration(ab6,1);
duration(m1,16); duration(m2,8); duration(m3,8); duration(m4,8);
duration(m5,8); duration(m6,20); duration(me,0); duration(l,2);
duration(t1,12); duration(t2,12); duration(t3,12); duration(t4,12);
duration(t5,12); duration(ua,10); duration(v1,15); duration(v2,10);
duration(k1,0); duration(k2,0); duration(pe,0);
```

```
precedence([ [pa,a1], [pa,a2], [pa,a3], [pa,a4], [pa,a5], [pa,a6],
[pa,ue], [a1,s1], [a2,s2], [a3,p1], [a4,p2], [a5,s5], [a6,s6], [p1,s3],
[p2,s4], [p1,k1], [p2,k1], [s1,b1], [s2,b2], [s3,b3], [s4,b4], [s5,b5],
[s6,b6], [b1,ab1], [b2,ab2], [b3,ab3], [b4,ab4], [b5,ab5], [b6,ab6],
[ab1,m1], [ab2,m2], [ab3,m3], [ab4,m4], [ab5,m5], [ab6,m6], [m1,t1],
[m2,t1], [m2,t2], [m3,t2], [m3,t3], [m4,t3], [m4,t4], [m5,t4], [m5,t5],
[m6,t5], [m1,me], [m2,me], [m3,me], [m4,me], [m5,me], [m6,me], [me,k2],
[l,t1], [l,t2], [l,t3], [l,t4], [l,t5], [t1,v1], [t5,v2], [v1,pe],
[t2,pe], [k2,pe], [t3,pe], [k1,pe], [ua,pe], [t4,pe], [v2,pe] ]);

max_nf([ [a1,s1,3], [a2,s2,3], [p1,s3,3], [p2,s4,3], [a5,s5,3],
[a6,s6,3] ]);

max_ef([ [s1,b1,4], [s2,b2,4], [s3,b3,4], [s4,b4,4], [s5,b5,4],
[s6,b6,4] ]);

min_nf([ [me,ua,-2] ]);

min_af([ [ue,s1,6], [ue,s2,6], [ue,s3,6], [ue,s4,6], [ue,s5,6],
[ue,s6,6] ]);

resource([ [excavator, [a1,a2,a3,a4,a5,a6]],
[pile_driver, [p1,p2]], [carpentry, [s1,s2,s3,s4,s5,s6]],
[concrete_mixer, [b1,b2,b3,b4,b5,b6]],
[bricklaying, [m1,m2,m3,m4,m5,m6]],
[crane, [l,t1,t2,t3,t4,t5]], [caterpillar,[v1,v2]] ]);

make_data(Lt, Ld) <-
      job(Lt) &
      mk(Lt,[],Ld);

mk([],Ld,Ld);
mk([J|L],CL,Ld) <-
      duration(J,D) &
      @Dtask(S) &
      append(CL, [[J,S,D]],NL) &
      mk(L,NL,Ld);

make_disj(R,Ld,Disj) <-
      resource(R) &
      md(R,Ld,[],Disj) ;

md([],Ld,CL,CL);
md([R|L],Ld,CL,Disj) <-
      md1(R,Ld,[],AL) &
      form_disj(AL,[],BL) &
      append(CL,BL,DL) &
      md(L,Ld,DL,Disj);

md1([R|[L]],Ld,CL,NL) <-
      md2(L,Ld,CL,NL);
```

```
md2([],_,NL,NL);
md2([H|L],Ld,CL,NL) <-
        X = [H,S,D] &
        delete(X,Ld,Ls) &
        append(CL,[X],AL) &
        md2(L,Ls,AL,NL);

form_disj([],NL,NL);
form_disj([[_,S,D]|L],CL,NL) <-
        fm1(L,S,D,[],AL) &
        append(CL,AL,BL) &
        form_disj(L,BL,NL);

fm1([],S,D,NL,NL);
fm1([[_,Sh,Dh]|L],S,D,CL,NL) <-
        append(CL,[[S,D,Sh,Dh]],BL) &
        fm1(L,S,D,BL,NL);

data_structure(Ld,Disj,End) <-
        make_data(Lt,Ld) &
        make_disj(R,Ld,Disj) &
        member([pe,End,_],Ld);

constraint(Ld) <-
        p_constraint(Ld) &
        d_constraint(Ld);

p_constraint(Ld) <-
        precedence(Lp) &
        pc1(Lp,Ld);

pc1([],Ld);
pc1([[Ti,Tj]|L],Ld) <-
        member([Ti,Si,Di],Ld) &
        member([Tj,Sj,_],Ld) &
        Sj >=@ Si + Di &
        pc1(L,Ld);

d_constraint(Ld) <-
        member([pa,Spa,_],Ld) &
        member([l,Sl,_],Ld) &
        Spa = 0 &
        Sl = 30 &
        pd1(Ld) &
        pd2(Ld) &
        pd3(Ld) &
        pd4(Ld) ;

pd1(Ld) <-
        max_nf(Lc) &
        pd1_aux(Lc,Ld);
```

```
pd1_aux([],Ld);
pd1_aux([[Ti,Tj,D]|L],Ld) <-
      member([Ti,Si,Di],Ld) &
      member([Tj,Sj,_],Ld) &
      D1 is  D  +  Di &
      Sj <==@ Si + D1 &
      pd1_aux(L,Ld);

pd2(Ld) <-
      min_af(Lc) &
      pd2_aux(Lc,Ld);

pd2_aux([],Ld);
pd2_aux([[Ti,Tj,D]|L],Ld) <-
      member([Ti,Si,_],Ld) &
      member([Tj,Sj,_],Ld) &
      Sj >==@ Si + D &
      pd2_aux(L,Ld);

pd3(Ld) <-
      max_ef(Lc) &
      pd3_aux(Lc,Ld);

pd3_aux([],Ld);
pd3_aux([[Ti,Tj,D]|L],Ld) <-
      member([Ti,Si,Di],Ld) &
      member([Tj,Sj,Dj],Ld) &
      D1 is D + Di - Dj &
      Sj <==@  Si + D1 &
      pd3_aux(L,Ld);

pd4(Ld) <-
      min_nf(Lc) &
      pd4_aux(Lc,Ld);

pd4_aux([],Ld);
pd4_aux([[Tj,Tj,D]|L],Ld) <-
      member([Ti,Si,Di],Ld) &
      member([Tj,Sj,Dj],Ld) &
      D1 is D + Di &
      Sj >==@ Si + D1 &
      pd4_aux(L,Ld);

choice(Disj,List,End) <-
      reverse(Disj,D) &
      disj_con(D) &
      labelling(List) ;

label_tl(L,NL) <-
      label_tl_aux(L,NL)  ;
label_tl_aux([],[]);
```

```
label_tl_aux([[_,S,_]|L],[S|Lv]) <-
      label_tl_aux(L,Lv);


disj_con([[Si,Di,Sj,Dj]|L]) <-
      disjunctive(Si,Di,Sj,Dj)  &
      disj_con(L);
disj_con([]);

disjunctive(Si,Di,Sj,Dj) <-
      Sj >==@ Si + Di  ;
disjunctive(Si,Di,Sj,Dj) <-
      Si >==@ Sj + Dj ;

bridge(Ld, End) <-
      data_structure(Ld,Disj,End) &
      constraint(Ld) &
      label_tl(Ld,List) &
      minimize_maximum(choice(Disj,List,End) , List) ;

labelling([X|Y]) <-
      deleteff(V,[X|Y],Ls) &
      indomain(V) &
      labelling(Ls);
labelling([]);
```

Appendix A5

A Mini-manual of WUP3.F

I) Introduction

WUP3.F employs a collection of consistency techniques(CTs) to solve discrete combinatorial problems, a class of constraint satisfaction problems. It is based on a conventional Prolog interpreter called WUP3.1; if CTs is not used, WUP3.F behaves the same way as WUP3.1. In this manual, the special features of WUP3.F are highlighted. For a more detail description of the system, please refer to body of this thesis.

II) Special Features

A. Finite Domains

In WUP3.F, a variable can be associated with a domain which is a finite set of natural numbers. This type of variables are called d-variables. They allow early detection of failures and facilitate the implementation of special inference rules. To use d-variables, a predicate, for example *pred*, must be submitted to a domain declaration of the following form:

domain @Dpred($a_1, \ldots, a_n$);

where each a_i can have one of the following values:

- h - the ith argument can be any term in the Herbrand universe;
- l .. u - the domain of the ith argument is a set of consecutive natural numbers with lower bound l and upper bound u;
- $[e_1, \ldots, e_k]$ - the domain of the ith argument is a set that contains all e_i's.

A domain declaration is usually included in the source code of a Prolog. When the code is consulted, the domain declaration is executed as a built-in predicate. Finite domains are seldom used independently; they are usually combined with other features of the interpreter.

B. Special Inference Rules

Two inference rules are implemented in WUP3.F: Forward Checking Inference Rule(FCIR) and Looking Ahead Inference Rule(LAIR). They allow the users to use either the forward checking approach or the looking ahead approach to solve a general constraint. In most cases, these rules are better alternatives than resolution. To use FCIR, a constraint must be submitted to a forward declaration of the form:

forward @Fpred($a_1, \ldots, a_n$);

where each a_i has the value of either 'g' or 'd'. A 'g' indicates that the corresponding argument in the constraint must be ground before the constraint can be executed. A 'd'

indicates that the argument must be a d-variable. FCIR is used to solve a constraint submitted to a forward declaration when there is only one uninstantiated d-variable.

To use LAIR, a constraint must be submitted to a lookahead declaration of the form:

lookahead @Lpred($a_1, \ldots, a_n$);

where each a_i is either a 'd' or a 'g', similar to that of a forward declaration. LAIR will be used to solve a constraint when all arguments that have 'g''s in the declaration are ground and at least one of the arguments that has a 'd' in the declaration is an uninstantiated variable with a finite domain. LAIR may not be able to solve the constraint completely. The constraint in this case will be delayed to wait for more information.

C. Built-in predicates

WUP3.F contains a collection of built-in predicates that implements CTs. Some of these predicates are used to instantiate d-variables. Some are implemented as specializations of the three inference rules (i.e. including the Partial Looking Ahead Inference Rule); and some employ the branch and bound method to solve combinatorial problems. The next section of the manual gives a list of all these predicates.

III) List of built-in predicates:

delay X

a delay declaration for predicate X;

deleteff(V,L,RL)
deleteffc(V,L,RL)

V is a d-var chosen form the list of d-vars L using the first fail principle. RL is the remaining list. deleteffc uses also the number of constraints in which V is involved as a selection criterion;

domain X

a domain declaration for predicate X;

element(I,L,E)

the Ith element of list L is E. Its lookahead declaration is element(d,g,d);

first_assertion(Head)

Head is the head of the first instance of an assertion whose head unifies with Head;

forward X

a forward declaration for predicate X;

indomain(X)

if X is a d-var, it will be instantiated with an element in its domain;if X is an integer, the predicate succeeds; otherwise, the predicate fails;

list_hidden(X)

this predicate lists the content of a hidden module which is either domain, forward, lookahead or delay;

lookahead X

a lookahead declaration for predicate X;

max@(X,Y,Z)

Z is the maximum of X and Z. Its lookahead declaration is max@(d,d,d);

minimize_maximum(T,L)

the branch and bound method is used to solve T so that the maximum element of L is minimized;

minimize(T,C)

the branch and bound method is used to solve T so that the value of C is minimized;

min@(X,Y,Z)

Z is the minimum of X and Z. Its lookahead declaration is min@(d,d,d);

reduce(X)

a linear equation or inequation is solved;

stats

print a snapshot of the runtime statistics of the current execution;

X <>@ Y + C

X is not equal to the sum of Y and C. Its forward declaration is d <>@ d + g;

X <>@ Y - C

X is not equal to Y minus C. Its forward declaration is d <>@ d - g;

X !=@ Y

X is not equal to Y. Its forward declaration is d !=@ d;

X <@ Y <- @FB_lt(X,Y);

X is less than Y. Its forward declaration is d <@ d;

X <=@ Y

X is less than or equal to Y. Its forward declaration is d <=@ d;

Each of the following predicates are expressed in term of one of the above predicates:

```
X >@ Y <- Y <@ X;
X >=@ Y <- Y <=@ X;
```

The following predicates are implemented specially for the N-queens problem and the scheduling problem:

X is@ Y + C

X is the sum of Y + C. Its forward declaration is d is@ d + g;

X >==@ Y +C

X is greater than or equal to the sum of Y + C. Its lookahead declaration is d >==@ d + C;

X <==@ Y + C

X is less than or equal to the sum of Y + C. Its lookahead declaration is d <==@ d + C;

IV) A Sample Session

The following is a sample session of using WUP3.F. In the session, the 5-queens problem is solved and All solutions are shown:

```
> wupF
Waterloo Unix Prolog [Version 3.F August 1989]
usr: ?consult(queen);
{}
{}
usr: ?cat(queen);
@Dqueen(X) <-
        safe(X) &
        labelling(X);

safe([]);
```

```
safe([F|T]) <-
        noattack(F,T) &
        safe(T);

labelling([]);
labelling([X|Y]) <-
        indomain(X) &
        labelling(Y);

noattack(X,Y) <-
        noattack(X,Y,1);

noattack(X,[],Nb);
noattack(X,[Y|YT],Nb) <-
        Y !=@ X &
        X <>@ Y + Nb &
        X <>@ Y - Nb &
        Nb1 is Nb + 1 &
        noattack(X,YT,Nb1);

% domain declaration for the @Dqueen predicate
?domain @Dqueen(1 .. 5);
{}
usr: ?@Dqueen([X1,X2,X3,X4,X5]);
{ X1=1, X2=3, X3=5, X4=2, X5=4 };

{ X1=1, X2=4, X3=2, X4=5, X5=3 };

{ X1=2, X2=4, X3=1, X4=3, X5=5 };

{ X1=2, X2=5, X3=3, X4=1, X5=4 };

{ X1=3, X2=1, X3=4, X4=2, X5=5 };

{ X1=3, X2=5, X3=2, X4=4, X5=1 };

{ X1=4, X2=1, X3=3, X4=5, X5=2 };

{ X1=4, X2=2, X3=5, X4=3, X5=1 };

{ X1=5, X2=2, X3=4, X4=1, X5=3 };

{ X1=5, X2=3, X3=1, X4=4, X5=2 };

no (more) answers
usr: ?quit;
>
```